Applications of Genetic Algorithms on theoretical fully-autonomous road systems

University of Birmingham

Sam Barrett, 1803086
sjb786@student.bham.ac.uk
Programme: MSci Computer Science
Supervisor: Dr Miriam Backens
Word count: 14349

May 7, 2021

Abstract

Efficiently routing vehicle traffic is fast becoming a key area of research. With the number of vehicles on the road increasing, existing infrastructure is under growing strain. Extending road networks is a costly process and requires ongoing, expensive maintenance.

Human drivers do not make efficient use of the existing road network, and so many approaches are being investigated for increasing this efficiency, thereby increasing throughput on the same series of roads.

In this paper I investigate the potential applications of Genetic Algorithms in this space. Building and evaluating a parallel cooperative coevolutionary algorithm (PCCGA), utilising Bézier curves to encode individuals in each sub-population.

I conclude that this approach has the potential, given more research and development time, to consistently produce feasible, safe routes for multiple agents across a network of roads.

Contents

List of Figures

1 Introduction

Over the past 100 years, roads have become more and more congested. The number of vehicles on the road in the UK alone has increased by more than 50% since 1994[1], this is thought to be considerably higher in developing countries, and the trend over the past decade is a 1-2% growth year-on-year. This can be seen in Figure 1.1. All the while, road networks in countries such as the UK are not being extended, putting the existing infrastructure under increasing strain.

On top of this, human controlled vehicles do not efficiently utilise the roads available, requiring additional space per vehicle due to factors such as *reaction time*, if we were able to remove these human elements it would be possible to greatly increase the throughput of the same amount of road infrastructure.

We can also consider the impact of autonomous navigation on road fatality rates. Over the past 10 years there have been an average of 1984 deaths per year on UK roads[3], with around a further 24,000 people seriously injured. Many of these incidents will have occurred due to human error, error which could be eliminated if the human factor were to be removed from vehicle navigation and routing and replaced with a robust, reliable automated system.

Many common approaches to solving this problem rely on directly replacing each human driver with an artificially intelligent system, examples of such systems include Tesla's autopilot or Google's *Waymo* taxi service. These are designed to plan and primarily consider a single vehicle. By taking this approach, these systems still have to account for this *human factor*, reducing much, if not all, efficiency gains.

In this paper, a more totalitarian system is proposed, designed and evaluated in which every agent is working to reduce the **net** travel time of the system, whilst strictly enforcing safety guarantees by avoiding collisions and infeasible space.

1.1 Goals

The overarching goal of this project is to optimally route traffic in the setting of a fully-autonomous road system. This however, is an aspirational goal with many sub-requirements to be fulfilled before it can come to fruition.

Here an *optimal* route is defined as the shortest possible path between two given points such that, it does not pass through infeasible space or collide with any other routes which may be executed concurrently. In reality, due to time and technical constraints, despite the Genetic Algorithms being

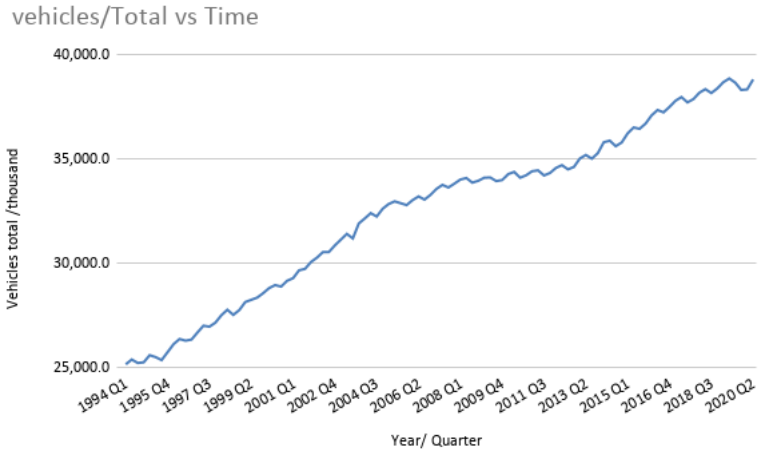


Figure 1.1: Total vehicles registered on UK roads over time[1]

probabilistically optimal, we will relax this requirement to be the *best*, or shortest route which still conforms to these constraints and is found within a certain number of generations over a population of a given size.

Initially, we want to be able to route a single agent through a single stretch of road, avoiding any and all infeasible space present within the road's boundaries whilst also staying within these boundaries at all times.

This will require the definition of an individual encoding in both the phenotypic and genotypic spaces (See Section 2.1) along with corresponding genetic operators which can operate on the individuals genotypic structure.

Building on this, the next sub-goal will be to create multiple routes with these characteristics whilst adding the requirement that none of them collide with each-other.

This will require a notion of velocity in order to accurately extrapolate the position along a route an agent is at a given point in time. This will also require the ability to detect if and where two routes intersect.

Finally, we need to wrap this functionality in such a way that allows for agents to navigate through multiple stretches of road, connected via intersections, with each sub-route maintaining all the requirements of the solutions to the previous sub-problems.

Formal definitions of these goals can be seen in Problems 1,2 and 3

1.1.1 Requirements

The goal of this project was not to produce a production-ready system, but instead, to investigate the plausibility of GAs on the real future possibility of completely autonomous road networks. As such I feel it useful to outline the theoretical requirements of a production grade system. To do this formally

Top-level Goal

Input:

- A road system $\mathcal{R}$ represented as a graph $\mathcal{R} = (V, E)$

  Where each edge $e \in E$ is a section of road defined as the space between two vertices $v^1, v^2 \in V$ which is bounded by two functions $b_1(x)$ and $b_2(x)$ and augmented by a set of obstacles O representing infeasible sections of road space

- a set of agents, $A = \{a_1, \ldots, a_n\}$
- a list of vertex pairs representing start and goal points for each agent:

  $\mathcal{V} = \left\{(v_1^1, v_1^2), \ldots, (v_n^1, v_n^2)\right\}, v_{i\in[1\ldots n]}^{j\in[1,2]} \in V$

  where the route for agent a_i is from vertex v_i^1 to v_i^2

Question: What is the set of optimal routes $\mathcal{V}_{optimal}$, where the i^{th} element is defined as a set of linked Bézier curves connecting v_i^1 and v_i^2 through feasible space?

Problem 1

Sub-Goal: Cooperative route planning

Input:

- A start point P_{start} and a goal point P_{goal}
- A section of road as defined in Problem 1
- A knowledge of all routes being planned to be executed concurrently.

Question: What is the optimal route, in the form of a Bézier curve, between these two points s.t. it does not collide with any other agents or pass through any infeasible regions in the road space?

Problem 2

Sub-Goal: Single agent route planning

Input:

- A start point P_{start} and a goal point P_{goal}
- A section of road as defined in Problem 1

Question: What is the optimal route, in the form of a Bézier curve, between these two points s.t. it does not pass through any infeasible regions in the road space?

Problem 3

I will employ Propositional and Temporal logic.

1. The system should never return a set of routes such that, for any time $t, \forall i \in [1, n], \forall j \in [1, n], j \neq i, I_i(t) = I_j(t)$. I.e. for any time, no routes should inhabit the same point, meaning there are no collisions in the planned routes. Where $I_i(t)$ is the position of agent i at time t and n is the number of agents being planned for.
2. The system should never return a set of routes such that, for any time $t, \forall i \in [1, n], \forall o \in O, I_i(t) \notin o$, I.e. for any time, no routes should pass through and infeasible space defined by any obstacles, $o \in O$.

2 Background

2.1 Genetic Algorithms

2.1.1 History

Genetic algorithms (GAs) are a type of meta-heuristic optimisation technique that employ the same rationale as classical evolution seen in nature.

Genetic Algorithms can trace their origins back to the late 1960s when they were first proposed by John Holland, though he then referred to them as *Genetic Plans*[1]. Holland went on to write the first book on the subject titled *Adaptation in Natural and Artificial Systems*[2] in 1975. The field did not find much reception with Holland stating in the preface to the 1992 rerun:

> *"When this book was originally published I was very optimistic, envisioning extensive reviews and a kind of 'best seller' in the realm of monographs. Alas! That did not happen."*

However, in the early nineties, Genetic algorithms surged in popularity along with the area of Artificially Intelligent planning as a whole, leading to Holland republishing his book and solidifying his position as the field's founder.

2.1.2 Definition

In a general sense, optimisation techniques work to find the set of parameters $\mathcal{P}$ that minimise an objective function $\mathcal{F}$. Genetic algorithms approach this by representing these sets as individuals in a population, P. Over the course of multiple generations, the best solutions are determined and promoted until termination criteria are met or the maximum number of generations is reached.

As our candidates are essentially a collection of parameters to the function we are trying to optimise, we can extend our metaphor further by mapping each element of a individual to a *gene* in a individual's genome.

The representation we use in a GA is problem-specific. Often we have to provide functions to facilitate the mapping between the problem-specific set of possible solutions and the encoded genotype space in which we optimise. The most basic representation being a string of binary numbers.

An individual's characteristics and genetic information is normally encapsulated within the Phenotype. Here not only the genetic information of

[1] This distinction was made to emphasise the *"centrality of computation in defining and implementing the plans"* [2]

the Genotype but also additional information, such as fitness, is stored in order to prevent it from being re-calculated as often.

In general, an individual's phenotype is the real-world representation of the candidate solution. The genotype is the low-level representation on which the genetic operators can operate. I will generally refer to both as an individual throughout this paper as the distinction is only necessary at the implementation level.

Genetic algorithms are both *probabilistically optimal* and *probabilistically complete*[4] meaning that: given infinite time, not only will the algorithm find *a* solution, (if one exists), it will find **the** optimal solution from the set of all possible solutions, $\mathcal{P}^*$.

Algorithm 1: Modern Generic Genetic Algorithm

Result: Best Solution, $p_{\texttt{best}}$
Generate initial population, P_0 of size n;
Evaluate fitness of each individual in P_0, $\{F(p_{0,1}, \ldots, p_{0,n})\}$;
while *termination criteria are not met* **do**
  Selection: Select individuals from P_t based on their fitness;
  Variation: Apply variation operators to parents from P_t to produce offspring;
  Evaluation: Evaluate the fitness of the newly bred individuals;
  Reproduction: Generate a new population P_{t+1} using individuals from P_t as well as the newly bred candidates.;
  t++
end
return $p_{\texttt{best}}$

As you can see from Figure 2.1 and Algorithm 1 the overall shape of GAs has not changed substantially over the course of the past 50 years. Being comprised of a series of operations that a starting population is piped through until criteria are met.

2.1.3 Genetic Operators

In the following section I will outline the various genetic operations that take place in a GA, they can be seen in Algorithm 1. Any operators used and analysed in Chapters 4 and 5 will have more detailed explanations of their process.

Selection

The selection procedure is the process by which the next generation of individuals is created from the current population. Individuals are selected relative to their fitness as determined by the Objective (fitness) function $\mathcal{F}$.

Some methods select only the best solutions by fitness. Others employ a more stochastic approach, such as roulette wheel selection, to increase

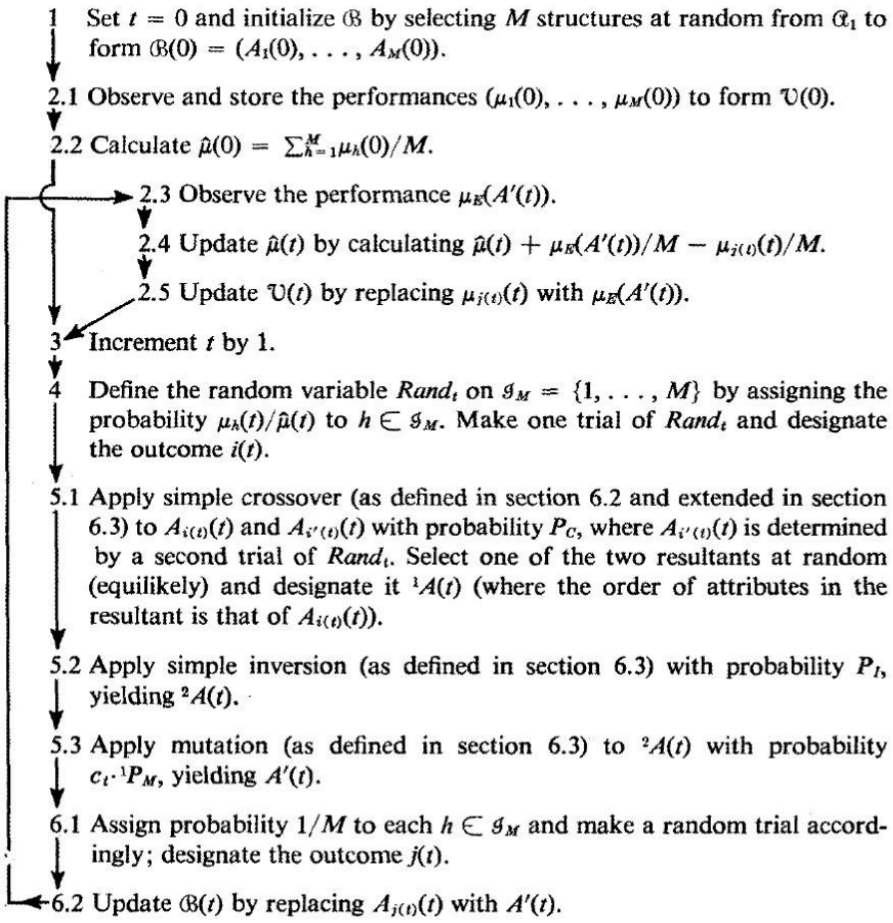


Figure 2.1: GA outlined in Holland's Original Book[2]

diversity and reduce complexity.

Fitness-Proportional Selection Fitness-proportional or Roulette wheel selection is a popular selection operator. It uses fitness to assign selection probability to each individual in a population.

The probability of selection for an individual i with fitness $\mathcal{F}(i)$ can be expressed mathematically as:

$$p_i = \frac{\mathcal{F}(i)}{\sum_{j=1}^{N} \mathcal{F}(j)} \tag{2.1}$$

Where N is the size of the population P

This is a simple approach but performs well and has very little performance overhead.

Ranked Selection Ranked selection is an alternative selection method, it works on the assumption that the individuals in your population are closely grouped in fitness. The main difference of this method when compared to fitness-proportional selection is that it ranks individuals based on relative fitness rather than absolute fitness.

The procedure can be written mathematically as:

Given a population P of size n ordered by fitness. We select the top γ-ranked individual, x_γ with a probability $p(\gamma)$. Where γ is the rank and $p(\gamma)$ is the ranking function.

The ranking function can take different forms including both linear and exponential ranking.

Linear ranking is defined as:

$$p(\gamma) = \frac{\alpha + (\beta - \alpha) \cdot \frac{\gamma}{n-1}}{n} \tag{2.2}$$

Where:

- $\sum_{\gamma=0}^{n} p(\gamma) = 1$
- $\alpha + \beta = 2$
- $1 \leq \beta \leq 2$

Variation

Variation in a GA is the process of altering the genome individuals to further explore the search space via stochastic local search. We perform this using two distinct sub-operations: Mutation and Crossover. Here we can view crossover as the *breeding* process and mutation as resembling the natural tendency for DNA to mutate over the course of generations.

Mutation In mutation we alter each gene with a set probability p_m known as the *mutation rate*. A standard value for a mutation rate is $\frac{1}{L}$ but it can fall anywhere in the range $p_m \in [\frac{1}{L}, \frac{1}{2}]$ where L is the length of the genome.

A low value for p_m a new individual which can be shown to be *close* to it's parents in the search space relative to their *Hamming distance*[2] if using a Binary coded GA. In real-coded GAs they can be shown to be close by the Euclidean distance between them.

In binary coded GAs we alter a given gene by *flipping* it's value. In real-coded GAs mutation operators include:

- Uniform Mutation
- Non-Uniform Mutation
- Gaussian Mutation

Uniform Mutation In uniform mutation, we select a parent p at random and replace a randomly selected gene $c_i \in p$ with a uniformly random number $c_i' \in [u_i, v_i]$ where u_i and v_i are set bounds.

Gaussian Mutation In Gaussian mutation, we select a parent p at random and randomly select a gene $c_i \in p$. We replace c_i with a new value c_i' which is calculated as follows:

$$c_i' = \min(\max(\mathcal{N}(c_i, \sigma_i), u_i), v_i) \tag{2.3}$$

Where $\mathcal{N}(c_i, \sigma_i)$ is a Gaussian distribution with a standard deviation σ_i and a mean of c_i. σ_i may depend on the length of the interval bound, $l_i = v_i - u_i$, typically (and in my implementation) $\sigma_i = \frac{1}{10} l_i$

Crossover

Crossover is a binary operation, taking two randomly selected *parents* from the population P_t with a probability $p_c \in [0, 1]$

There are two major forms of crossover for binary coded GAs: $n > 1$ point crossover and Uniform crossover.

For real-coded GAs you instead have a selection of Crossover operators including:

- Flat crossover
- Simple crossover
- Whole arithmetic crossover

[2]Hamming Distance: A metric for comparing two binary data strings. The Hamming distance between two strings is the number of bit position in which they differ.

- Local arithmetic crossover
- Single arithmetic crossover
- BLX-α crossover

Simple (one point) Crossover For simple crossover, randomly select a crossover point, $i \in \{1, \ldots, n\}$. All values before this point are swapped between the two parents. For 2 parents, $p_1 = \left\{x_1^{[1]}, \ldots, x_n^{[1]}\right\}$ and $p_2 = \left\{x_1^{[2]}, \ldots, x_n^{[2]}\right\}$ this can be represented as:

$$p_1' = \left\{x_1^{[1]}, x_2^{[1]}, \ldots, x_i^{[1]}, x_{i+1}^{[2]}, \ldots, x_n^{[2]}\right\} \tag{2.4}$$

$$p_2' = \left\{x_1^{[2]}, x_2^{[2]}, \ldots, x_i^{[2]}, x_{i+1}^{[1]}, \ldots, x_n^{[1]}\right\} \tag{2.5}$$

***k*-point Crossover** k-point crossover is an extension of one point crossover, instead of swapping genetic information about a single point, a random number k is chosen and information is swapped about k random points along the genome.

For the same two parents defined above, with k crossover points, $cp_1, \ldots, cp_k$ we can define our offspring as follows:

$$p_1' = \{x_1^{[1]}, \ldots, x_{cp_1-1}^{[1]}, \ldots, x_{cp_1}^{[2]}, \ldots, x_{cp_2-1}^{[2]}, \ldots x_{cp_k-1}^{[2-k\%2]}, \ldots, x_n^{[k\%2+1]}\} \tag{2.6}$$

$$p_2' = \{x_1^{[2]}, \ldots, x_{cp_1-1}^{[2]}, \ldots, x_{cp_1}^{[1]}, \ldots, x_{cp_2-1}^{[1]}, \ldots x_{cp_k-1}^{[k\%2+1]}, \ldots, x_n^{[2-k\%2]}\} \tag{2.7}$$

In cases where $k > 1$ and % is the modulo operator.

Evaluation

After developing potential new individuals, the fitness of these new individuals is calculated using the objective function, $\mathcal{F}$

Reproduction

Here the next generation of individuals is constructed. There are multiple potential methods employed here with the simplest being to just replace the least fit individuals with additional copies of the fitter individuals, be they pre-existing or newly generated.

In this stage various heuristics or alternative strategies can be implemented to speed up or slow down the convergence rate. Whilst their fitness may be low, having a diverse population allows for more of the search space to be explored. If the algorithm converges too quickly it may get *stuck* in local minima.

2.2 Bézier Curves

In my implementation, I utilise Bézier curves to encode complex route arcs between a series of points. Here I will briefly outline their construction and mathematical basis.

Bézier curves were popularised by and named after French auto-body[3] designer Pierre Bézier in the 1960s and are commonly found in computer graphics today. They are parametric curves made up of a series of *control points* which contort the shape of a line to produce a curve of nearly any shape. Although their mathematical basis was established in 1912 by Sergei Bernstein in the form of Bernstein polynomials[5].

A Bézier curve is said to have a degree of $n-1$ where n is the number of control points including the start and end point of the curve.

2.2.1 Formal Definition

Bézier curves can be simply and explicitly defined for degrees of 1 through 4, however, the general case of n points is more useful to us. In their general form they can be defined recursively or explicitly. I implement the recursive version as it is much more readable for a programming language that supports recursion.

They are defined recursively as follows:

$$\mathbf{B}_{P_0}(t) = P_0 \tag{2.8}$$

$$\mathbf{B}_{P_0,P_1,\ldots,P_n}(t) = (1-t)\mathbf{B}_{P_0,P_1,\ldots,P_{n-1}}(t) + t\mathbf{B}_{P_1,P_2,\ldots,P_n}(t) \tag{2.9}$$

Where the initial curve $B = B_{P_0,P_1,\ldots,P_n}$ and the parameter t is a real number in the range of $[0,1]$. Therefore, the smoothness of the curve is determined by the granularity of the parameter when realising the curve.

An example of how the parameter t corresponds to points on the curve can be seen in Figure 2.2, the red numbers indicate the value for t that corresponds to the green points on the curve and the purple points correspond to the control points. I.e. $B(0.37)$ produces the coordinates for the first labelled point seen, where the curve shown is B.

2.2.2 Length of a Bézier curve

The length of a Bézier curve is a non-trivial calculation and is best approximated as:

$$L = \frac{(2L_c + (n-1)L_p)}{n+1} \tag{2.10}$$

[3] so I feel it is quite fitting that they find use in 21^{st} century automotive problems

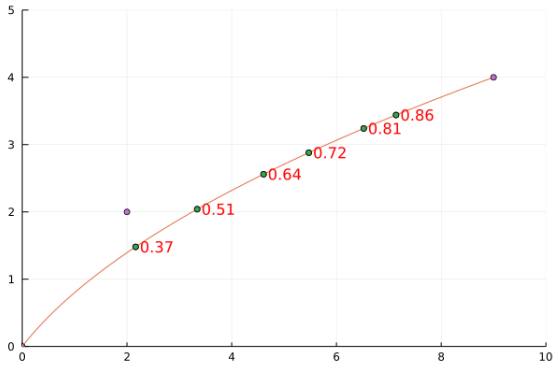


Figure 2.2: Example of a Bézier curve with a random sample of points

Where L_c is chord length and L_p is polygon length. This was proven by Jens Gravesen[6] in 1997.

Chord length is given by:

$$L_c = \sqrt{(P_{0_x} - P_{n_x})^2 + (P_{0_y} - P_{n_y})^2} \tag{2.11}$$

and polygon length by:

$$L_p = \sum_{i=0}^{n-1} \sqrt{(P_{i_x} - P_{i+1_x})^2 + (P_{i_y} - P_{i+1_y})^2} \tag{2.12}$$

Or, alternatively:

$$L_p = \sum_{i=0}^{n-1} L_c(B_{P_i,P_{i+1}}) \tag{2.13}$$

2.2.3 Subdivision of n-degree Bézier curves

Subdivision of Bézier curves is a key stage in the process of finding any intersections between two Bézier curves. The process of subdivision is performed using De Casteljau's algorithm. This algorithm was developed by Paul de Casteljau while working at Citroen in 1959 to work with the family of curves that would later be formalised into Bézier curves by Pierre Bézier[4].

It is a simple procedure, given a Bézier curve B, it can be split at any arbitrary point $t \in [0, 1]$ into two curves with control points of the form:

$$\beta_1 = B_{P_0}(t), B_{P_0,P_1}(t), \ldots, B_{P_0,\ldots,P_n}(t) \tag{2.14}$$

$$\beta_2 = B_{P_n}(1-t), B_{P_n,P_{n-1}}(1-t), \ldots, B_{P_n,\ldots,P_0}(1-t) \tag{2.15}$$

[4]Bézier curves have also been called de Casteljau curves as much of his work preceded that of Pierre Bézier's

2.3 Fully Autonomous Road Networks

Fully autonomous road networks are by no means a novel concept. They have appeared in fiction since the mid 20th century and have been a real possibility for the past decade. In a fully autonomous road networks *(FARN)*s, all vehicles are self-operating with their routing either being determined on a vehicle-by-vehicle *selfish* basis or by a larger system managing routes for all nearby agents.

In the latter system protocols that promote the net increase in efficiency can be implemented. However, this sort of system will require a form of government mandate as a universal routing protocol would need to be established and implemented by all auto manufacturers; this is no small undertaking.

2.4 Miscellaneous

2.4.1 Bounded Boxes

A bounded box of a curve is defined as the box constructed from the points $(x_{min}, y_{min}), (x_{min}, y_{max}), (x_{max}, y_{min}), (x_{max}, y_{max})$ Where $x_{min,max}$ and $y_{min,max}$ are the minimum and maximum x and y values from all control points respectively. An example can be seen in Figure 2.3

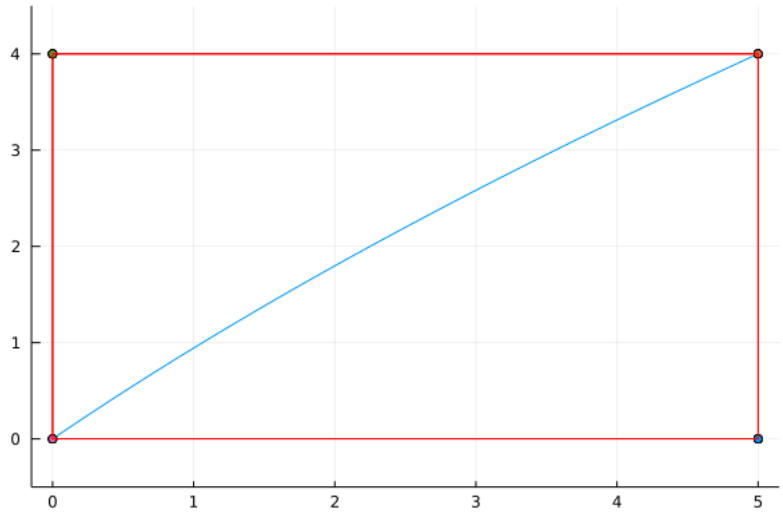


Figure 2.3: A bounding box for a Bézier curve of degree 2

3 Literature Review

3.1 Genetic algorithms for route generation

Classical Genetic algorithms have seen research and applications in a number of fields over the past 40 years ranging from Heart disease diagnosis[7] to predicting the strength of concrete under various conditions[8].

There is a substantial and growing body of research specifically looking at the applications of GAs in route planning for autonomous agents, however, many are focusing on either abstract robotic agent planning in discrete search spaces or failing to take into account the wider environment of multiple agents.

3.1.1 Genetic algorithms for cooperative route generation

Cai & Peng - Cooperative Coevolutionary Adaptive Genetic Algorithm in Path Planning of Cooperative Multi-Mobile Robot Systems[9]
In this 2001 paper, Cai & Peng give an example of a Cooperative Coevolutionary Adaptive Genetic Algorithm (CCAGA) for planning routes for multiple autonomous agents.

Their approach features a discrete grid-based search space featuring rectangular obstacles which the routes must avoid. Their chromosomes are encoded using real-values representing the x and y coordinates of the search grid.

Their fitness function has two stages, in the first generation the fitness of a candidate solution is purely determined by the global *optimality* of the route. In all subsequent generations the fitness also takes into account the best routes from each of the other sub-populations.

They conclude that their approach yields good robustness and convergence as well as being suitable for parallel execution, allowing for systems employing this GA approach to run very efficiently.

Elshamli et al. - Genetic algorithm for dynamic path planning [10]

In this 2004 paper Elshamli et al. propose a Genetic Algorithm Planner (GAP) which generates routes made up from edges connecting n points. They penalise routes for colliding with obstacles as well as for not being *smooth*, the core of their route fitness is based on route length.

The smoothness metric is based on the angles between the edges as they connect at a node.

They do not provide concrete evaluations for their implementation but seem to focus on the notion of *dynamic* search spaces and which of their approaches handle these best. They define their dynamic search spaces as spaces in which obstacles are added during generation, for example, after the first generation. Their proposed solutions to this are as follows:

- Utilise high mutation rate of 50% and a low crossover rate of 50%
- Introduce newly randomly generated individuals at the beginning of each generation
- A method by which to remember the fittest individuals at the end of each generation and to reintroduce them with a certain chance
- A combination of the previous two

In their testing the best average solution was achieved using the first of these approaches and the best overall solutions were achieved using the final solution.

All of these approaches could potentially be utilised in a cooperative, multi-agent, approach if we consider other agents to be dynamic obstacles.

Raul Kala - On-Road Intelligent Vehicles: Optimization Based Planning 2016[11] In his book, Kala proposes a GA as an embedded component of a larger planning system which also relies on Djiksra's algorithm for macro-level route planning along with the classical autonomous vehicle real-time collision avoidance toolbox (radar, sonar, etc.). In order to achieve a system capable of planning for multiple agents concurrently, Kala proposed the use of these real-time collision avoidance tools along with embedding *traffic rules* as heuristics into the Djiksta-based component. These traffic rules included "driving on the left" and "overtaking on the right" etc. This proposed system, while similar to mine, differs in scope. I propose a system for planning routes for a set of autonomous agents within a fully-autonomous road system, meaning agents will not (in theory) interact or encounter any other vehicles which are not a part of the planning network.

Kala also proposes a seemingly novel *road space coordinate system* which allows him to formally bound the genotypic search space by the dimensions of the road, eliminating the possibility of plotting routes outside of the road space. He goes on to show that this approach in his implementation greatly reduces the number of agents required to generate feasible solutions, though it is not discussed how this may impair the GAs ability to generate complex routes which may require control points to lie outside of the road space whilst still defining a feasible curve.

Cruz-Piris et al. - Automated Optimsization of Intersections Using a Genetic Algorithm[12] A paper published by Cruz-Piris et al. investigates the applications of (binary coded) Genetic Algorithms on routing traffic through busy intersections. Their proposed system is shown to have improvements in traffic throughput of anywhere from 9% to 36%.

Their approach involved several assumptions, many of which my research shares. Namely, they assumed vehicles do not stop at any point and that their speed remains constant at all points in time.

Their approach to modelling the solution space takes advantage of the fairly uniform and regular structure of large intersections, and although they work on modelling irregularly shaped intersections, this traffic cellular automata (TCA) model begins to fall apart when the size of each cell cannot contain a vehicle and requisite surrounding space, or if input/output lanes do not lie in a regular grid pattern. Facilitating irregularly shaped intersections seems to be a manual process, not a particularly viable method when you consider the number of unique intersections across the world or even just the Continental United States!

This approach seems to work relatively well for throughput optimisation problems such as junctions but does not scale well for less structured scenarios where all origins and goals cannot be known beforehand and are not necessarily the same for all subsequent runs.

Roberge et al. - Fast Genetic Algorithm Path Planner for Fixed-Wing Military UAV Using GPU[13] In this 2018 paper an interesting proposition was made, greatly increasing efficiency and viability of real-time planning via GAs through massively parallelising its core processes to run on GPUs. Their implementation shows a 290x times speedup compared with conventional sequential CPU execution. GAs lend themselves quite nicely to parallel execution as we are applying the same operators to multiple individuals of the same shape.

3.2 Bézier curves for route generation

Chen et al. - Quartic Bézier Curve based Trajectory Generation for Autonomous Vehicles with Curvature and Velocity Constraints In this 2014 paper, Chen et. al research the efficacy of quartic (4-degree) Bézier curves for planning continuous routes between points for autonomous vehicles. In order to find an optimal solution they employ sequential quadratic programming.

In their approach Chen et al. also take into account the orientation, change in orientation and velocity of the routes generated in order to assure all generated routes are *safe* for occupied vehicles to follow.

This approach appears to be extremely computationally efficient on the

toy examples provided in the paper, on relatively low performance hardware 560 iterations can be computed in around 500ms!

There is no mention, however, to a cooperative element. Explicitly solving quadratic programming tasks is known to be **NP**-complete as shown by Vavasis in 1990[14] and adding onto that the requirement that all solutions interact nicely will only increase the real-world runtime of any solution. A real-world scenario with hundreds of vehicles may suffer from an incredibly high runtime, hardware requirement or both.

3.3 Fully Autonomous Road Networks

Fully Autonomous Road Networks (*FARN*s) have the possibility of improving travel in a number of different areas.

They have the potential of greatly reducing travel times by efficiently routing all vehicles with the goal of reducing the net travel time. Such a system would also be able to respond much faster than human drivers, allowing for large increases in permissible vehicle velocities thus, further increasing efficiency.

By extension of their higher efficiency, *FARN*s will also have a marked effect on vehicular energy consumption. The potential effects are summarised nicely by Vahidi and Sciarretta[15] where they show the use of vehicle automation could cause anywhere from a halving of "energy use and greenhouse gas emission" in an "optimistic scenario" to doubling them depending on the effects that present themselves such as reduced public transport usage. The increase in highway speed whilst increasing travel efficiency is predicted by Brown et al.[16] and Wadud et al.[17] as increasing energy use by anywhere from 5% to 30%.

4 Approach

4.1 Single Agent Approach

The goal of this section was to design a system that fulfilled the requirements outlined in Problem 3

The general shape of a GA as seen in Algorithm 1 is the same for almost all problems. But each individual operator implementation is domain specific.

4.1.1 Individual Encoding

As such, I first had to implement a method to generate an initial population. In order to do this, I had to define my programmatic representations of the phenotypes and genotypes of my individuals. I decided to base my pheno and genotypic structure on the that described by Kala[4].

In this approach, the phenotype is abstractly represented as a Bézier curve (Section 2.2). This is more concretely translated into a genotype encoded as a real-valued string, interleaving the x and y coordinates of each control point. Taking the form:

P_{0_x}	P_{0_y}	P_{1_x}	P_{1_y}	$\cdots$	P_{n_x}	P_{n_y}

Figure 4.1: Individual genotype as real-valued string

In the above form the descriptions of genetic operators found in Section 2.1.3 should be sufficiently detailed to re-implement.

Initially my Phenotype contained no additional information. However, I still created the distinction to allow for additional properties to be encapsulated during development.

4.1.2 Population Generation

Population generation theoretically can be very simple and generate very little diversity, relying on the other operators to explore the search space. However, the more diverse a set of initial routes one can generate, the quicker the system will converge to an optimal solution.

As such we want to ensure that both the number of control points as well as their position are properly varied. Therefore, my generation procedure randomly generated x position within the range of the start and goal x positions. The y positions can fall outside of the road boundaries as sometimes

this is necessary to properly guide a route around an infeasible region. All numbers in this process were generated using a uniform distribution so as maximise the amount of the search space that initial set of candidates covers.

The first and last control points **must** be the starting and goal position for the agent. In a Bézier curve these are the only two points that the curve definitely passes through. We must also ensure that the x coordinates of the control points are in order, so as to avoid routes which *double back* on themselves. This can also occur as the result of other operators such as crossover and mutation, we can fix this using a simple repair operator which re-orders points based on their x positions.

Once the initial population is generated, the main algorithmic loop can begin, this is the repeated application of the selection, crossover, mutation and evaluation operators (See Section 2.1.3) until set termination criteria are met or the maximum number of generations is completed, seen in Algorithm 1.

4.1.3 Fitness assessment

The foundation of my Fitness function is the length of each route, i.e. the length of a given n degree Bézier curve. This calculation can be accurately approximated using Equation 2.11 as the true calculation is expensive and does not generalise nicely to n degrees[6].

Minimising the length of a route is a good way of ensuring generated routes are direct, however, other values and heuristics are required to maintain other requirements such as to make sure changes in direction are not too severe or that a route does not pass through infeasible space (avoiding obstacles etc.).

Road Section Definition

I initially experimented with an alternative coordinate space which enabled me to completely enclose the search space within the area of the road, eliminating the possibility of generating routes that left the *road*. This was inspired by the "Road coordinate axis system" described by Kala in [11]. However I abandoned this concept as it did not appear to increase performance and introduced overhead when visualising or generating the routes.

Instead I defined each section of road using the following Structure:

```
struct Road
    boundary_1::Function
    boundary_2::Function
    obstacles::Array{Obstacle}
    length::Real
end
```

This approach allows me to define the upper and lower boundary of a road as any function, straight roads can be defined as a pair of straight lines a certain width apart, and arbitrarily complex roads can be represented using arbitrarily complicated functions.

For example the road seen in Figure 4.2 had boundary functions:

$$b_1(x) = 0$$
$$b_2(x) = 5$$

And the curved road section seen in Figure 4.3 functions:

$$b_1(x) = 2\cosh(0.1x) - 2$$
$$b_2(x) = 2\cosh(0.12x) + 8$$

Where in both cases the length of the road (and therefore upper bound of x) was 25, the lower bound of x was 0.

Note: this format of figure will be used throughout this report and shows a top-down view of the road space, the y axis corresponds to the road width and the x axis to the road length.

Figure 4.2: A straight road

Road Obstacles I have implemented two basic `Obstacle` types (`Rectangle` and `Circle`) allowing me to create regions of infeasible space within the valid road-space.

Infeasible space is then calculated by taking a sample of 500 points along a given curve B and checking which (if any) lie within any of the obstacles.

A Bézier curve is then constructed using two applications of De Casteljau's algorithm (See Section 2.2.3) using values for t derived from the sub-sample points. An illustration of this process can be seen in Figure 4.4. The length of this curve is weighted and added to the overall fitness of the candidate solution. The stages progress as Follows:

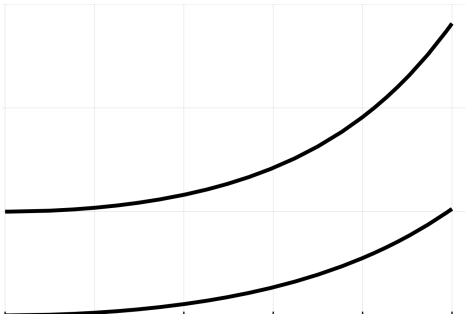


Figure 4.3: A curved road

1. Curve and obstacle drawn
2. Uniform sample of points chosen along curve
3. Curve segment from origin to first infeasible point isolated
4. Curve segment from last infeasible point to end position isolated
5. Infeasible distance calculated as difference in length between original curve and two isolated segments

A similar process is performed to determine how much of a curve passes too closely to an infeasible region, this is penalised with a separate, lower, weighting.

Now with the ability to calculate the length of a route as well as the distance it travels through/ too close to infeasible space, I can define my Fitness function:

$$\mathcal{F}(i) = \begin{array}{c} L_i + \\ \alpha \cdot \texttt{infeasible distance}(i) + \beta \cdot \texttt{high-proximity distance}(i) \end{array} \quad (4.1)$$

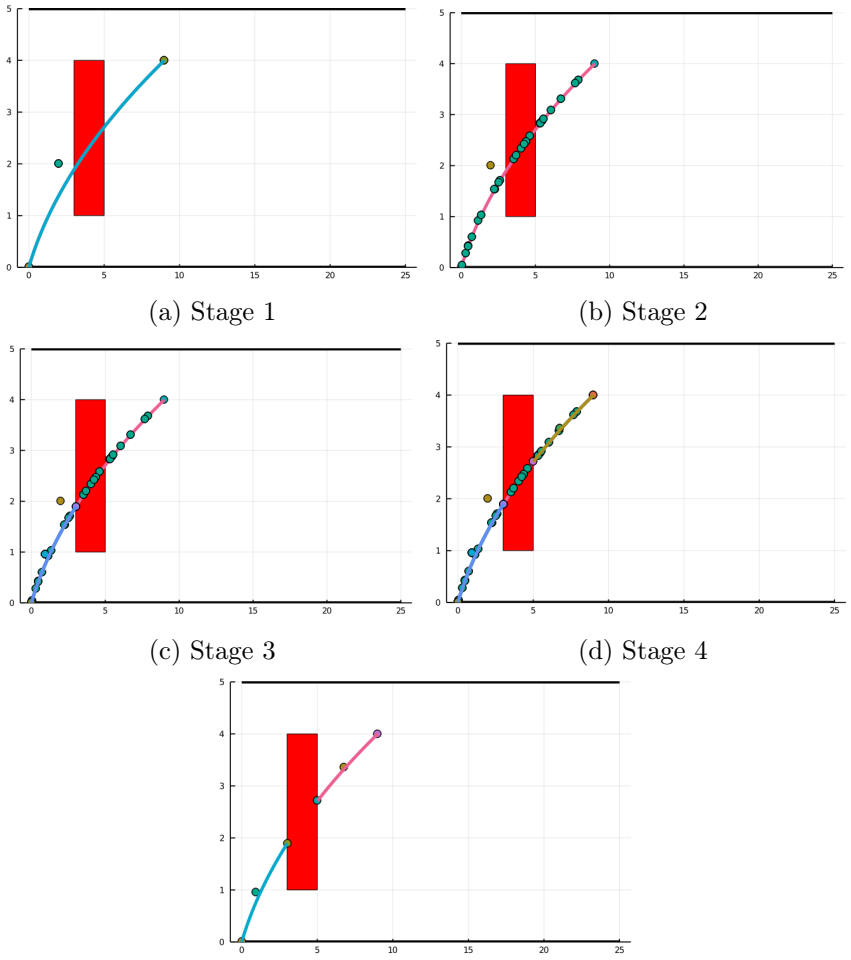
Where:

- L_i is the length of candidate i's route
- α and β are weighting parameters s.t.

  $\alpha > \beta > 1$

  therefore penalising infeasible distance harder than distance which is *too close* to infeasible space, guiding the GA to prioritise avoiding infeasible space.

This function is inspired by Kala[11] and proves to be a good basis for a naive single agent planner, futher augmentation is required to support multiple agents or variable velocity profiles.



(a) Stage 1

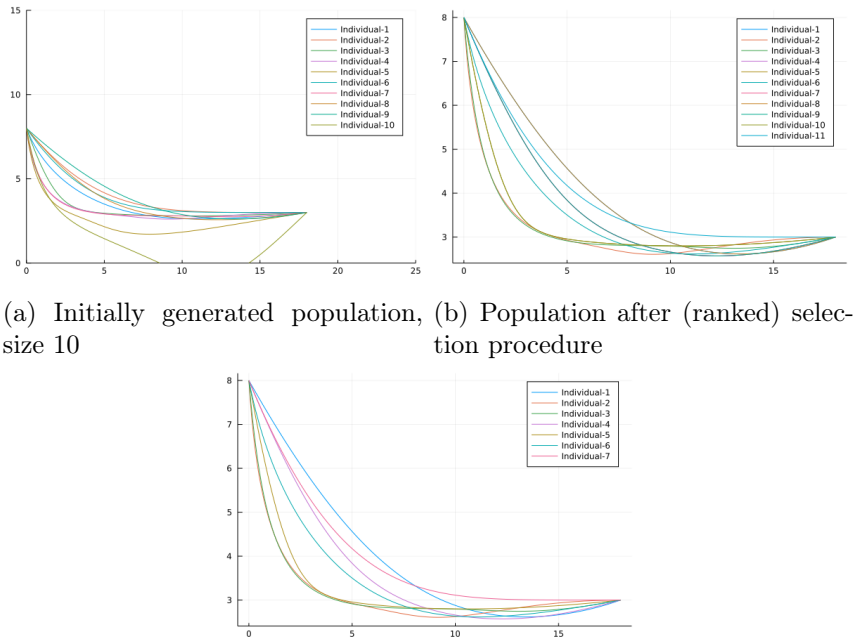
(b) Stage 2

(c) Stage 3

(d) Stage 4

(e) Stage 5

Figure 4.4: Illustration of finding infeasible section of curve



(a) Initially generated population, size 10

(b) Population after (ranked) selection procedure

(c) Population after (ranked) selection procedure, duplicates removed

Figure 4.5: Example of ranked selection

4.1.4 Genetic Operators

The abstract goal of each genetic operator has already been discussed in Section 2.1.3. Here I will discuss their concrete implementation with respect to my task.

Selection

My selection procedure follows very closely to the method outlined in Section 2.1.3, relying on the fitness function to duplicate fit individuals, replacing the less fit ones with a certain chance or reasoning.

An example of ranked selection on a sample population can be seen in Figure 4.5

Mutation

The variants of the mutation operator that I have implemented are Uniform and Gaussian. Both attempt to achieve the same effect via different methods.

Abstractly, my mutation operators select a control point with a random probability and mutate within fixed bounds. These bounds are tied to the size and shape of the road.

In my best performing operator, the Gaussian mutation operator, the range of x values for the mutation of a randomly selected point $p \in B = \{p_1, \ldots, p_n\}, n \in \mathbb{Z}^{\geq 2}$ are:

$$\mathcal{X} = [p_{1_x}, p_{n_x}], p_1, p_n \in B \tag{4.2}$$

I.e. a mutated control point cannot lie before or after the start or end points respectively. The range of y values a mutated point can take is within:

$$\mathcal{Y} = \left[-1.3 \cdot \frac{b_1(\mathcal{X}_1) + b_2(\mathcal{X}_2)}{2}, 1.3 \cdot \frac{b_2(\mathcal{X}_1) + b_2(\mathcal{X}_2)}{2}\right] \tag{4.3}$$

Meaning -1.3 times the average y coordinate of the bottom road boundary to 1.3 times the average of the top road boundary across the range of permitted x values. ± 1.3 was used after experimenting with values ranging from ± 2 to ± 0.5, 1.3 offers the procedure the chance to significantly affect the course of a route whilst, minimising the chance that the mutated route will be infeasible.

A pair standard deviations as previously outlined in the Background section are defined as:

$$\sigma_x = \frac{\mathcal{X}_1 - \mathcal{X}_2}{10} \tag{4.4}$$

$$\sigma_y = \frac{\mathcal{Y}_1 - \mathcal{Y}_2}{10} \tag{4.5}$$

Where $\mathcal{X}_1$ is the 1st element in $\mathcal{X}$, and the same notation applies for $\mathcal{Y}$
A new point, p', is then constructed as follows:

$$p'_x = \min(\max(\mathcal{N}(p_x, \sigma_x), \mathcal{X}_1), \mathcal{X}_2) \tag{4.6}$$
$$p'_y = \min(\max(\mathcal{N}(p_y, \sigma_y), \mathcal{Y}_1), \mathcal{Y}_2) \tag{4.7}$$

Point $p \in B$ is then replaced with our new point $p' = (p'_x, p'_y)$.

The result of the operation being applied to a example curve can be seen in Figure 4.6

Uniform mutation used the same bounds $\mathcal{X}, \mathcal{Y}$ as defined above, where the set bounds, u_i, v_i mentioned in Section 2.1.3 are correspond to the two values in each of $\mathcal{X}$ and $\mathcal{Y}$.

Crossover

Simple crossover is implemented as it is described in Section 2.1.3, two parents are randomly selected and exchange genotypic information about a random point. I have been careful not to choose a crossover point which splits a control point. The result of this operation is appended to the current

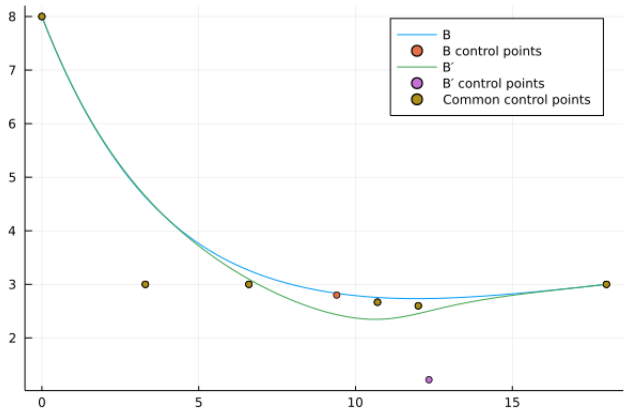
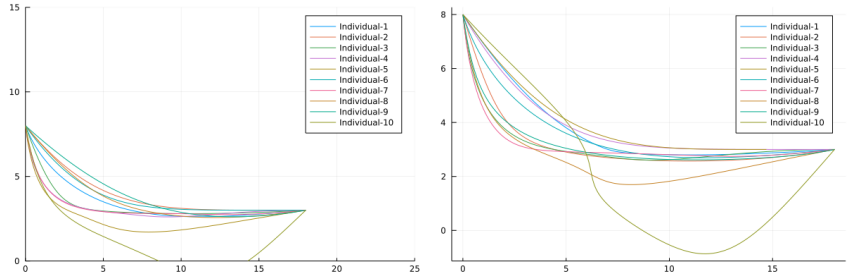


Figure 4.6: Example of Gaussian mutation on a Bézier curve



(a) Initially generated population, size 10

(b) Population after crossover operation is applied

Figure 4.7: Example of simple crossover

population, only the top n individuals are carried forward to the next generation. Once used to create offspring, an individual cannot again participate in crossover, however, duplicate individuals created via the selection process may effectively reproduce multiple times.

The result of a single (simple) crossover operation can be seen in Figure 4.7

k-point crossover was also implemented and follows much the same logic. However, here as described in Section 2.1.3, genotypic information is exchanged about k randomly selected points where k itself is randomly selected from the range $[1, \min(m, n)]$ where m is the number of control points in $parent_1$ and n is the number of control points in $parent_2$.

Each of the operators outlined in this section are then applied in the order shown in Algorithm 1, until the termination criteria are met. In this case my only termination criteria was the maximum number of generations had taken place.

4.2 Multi-agent Approach

So far we have only concerned ourselves with planning a route through a road-space for a single agent. However, in the real world, roads are seldom occupied by a single vehicle and as such we must consider how to efficiently plan a set of routes for a set of agents between between a set of coordinate pairs, such that, our agents do not collide at any point in time.

There are different approaches we can take to this problem.

My initial approach to a collaborative planning system was somewhat inefficient. The system operated by planning routes for each agent sequentially, using a growing *context* to keep track of the routes that had already been finalised. The first agent to be planned was not concerned with avoiding collisions, because, as far as it was concerned, there were no other agents in the system. Each subsequent agent checked if any of its candidate solutions intersected with any of the routes planned before, a very high penalty was applied to any routes with collisions.

This system was very inefficient with the final routes requiring a large number of checks to be carried out by my `bezInt` function, which itself suffered from high time (and space) complexity.

This system was improved upon by splitting the process over multiple threads, each agent being given a separate thread, communicating their most up-to-date plans via a shared array, S. Agent i will store its current fittest route in $S[i]$, this will be updated at the end of each generation over the i^{th} subpopulation. The calculation for fitness of any candidate checks for collisions with each route in S. This system may conduct more checks but it removes the problem of prioritising the first route to be planned and reduces the overall runtime of the system approximately proportionally to the number of threads available, an abstract view of the topology can be seen in Figure 4.8 with each agent $a_k, k \in [1, n]$ being mapped to a separate threaded instance of my GA planner. Each instance has access to the shared array which stores the fittest route after generation $i \in [1, m]$ with m being the maximum number of generations for each agent.

Updates to the shared array happen as and when each thread reaches the end of a generation. This can mean that routes that terminate before others have to adapt less to other routes, but this trade-off cannot be avoided without instituting constant communication between threads, which in and of itself introduces overhead and slows down planning.

This approach was inspired by that of Cai & Peng[9] in which they also implemented cooperative coevolutionary genetic algorithms by sharing the fittest solution of each subpopulation with the other subpopulations. My approach takes this a stage further through the use of threading, allowing for this sharing process to occur concurrently.

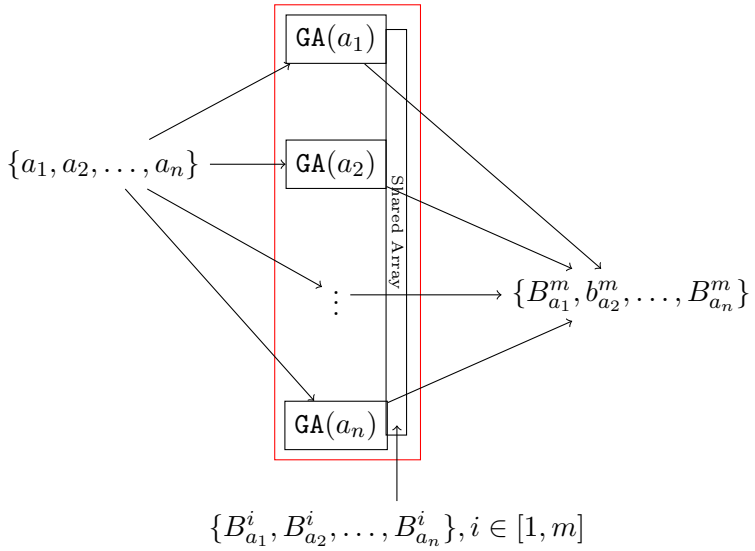


Figure 4.8: Parallel Cooperative Genetic Algorithm abstract topology

4.2.1 Collision Detection

Collision detection is one of the core obstacles to a viable cooperative route planner. You must be able to certify that your resulting set of routes do not collide at any point in time, else the entire system fails.

Simply detecting intersections is not enough as two routes can intersect but only collide if they pass through the same point at the same time. We therefore require some notion of time when realising our Bézier curve routes.

I made the assumption that each agent in my system travels at a uniform constant speed. To remove this assumption we would need to include a velocity-time profile in the genotype of each individual, this is an area for further research.

With this assumption, we can now say that two routes collide if they intersect and the distance from their respective origins and the point of intersection is the same (or below a threshold). We have now reduced this problem to finding a point of intersection between two Bézier curves, this is still a non-trivial task.

Bézier Curve intersection

The commonly employed technique for finding intersections between two n-degree Bézier curves is repeated recursive subdivision. Whilst it is possible to precisely calculate the point of intersection it is very difficult and computationally expensive and does not generalise well to n degrees. For example, to precisely find the intersection two curves of degree 3 we must solve a 9^{th} order equation, producing unstable results. This is described by Bézout's

theorem which states that two planar algebraic curves of degrees d_1 and d_2 will have up to $d_1 d_2$ intersection points.

In their 2006 paper Yap[18] proposed an efficient method for finding the intersection of two Bézier curves through a 2 stage process surrounding repeated subdivision.

During development I attempted to use a Bézier curve library (`libbezier`)[19] written for python but featuring a core written in Fortran, providing a C ABI. Julia, featuring interoperability with C and Fortran, allowed me to make direct calls to the Fortran subroutines and C functions. Ultimately, however, I encountered too many issues stemming from this library's inability to be run in parallel along with seemingly random segfaults which I struggled to diagnose through two levels of language abstraction (Fortran $\rightarrow$ C $\rightarrow$ Julia); this led me to write my own functions natively in Julia, although replicating the same accuracy as the nearly 4000 lines of Fortran proved challenging.

In order to speedup my Bézier intersection function I implemented a number of time saving measures:

1. If the convex hulls of two curves B_1 and B_2 intersect, i.e. some amount of the area of the convex hull of B_1 ($CH(B_1)$), lies inside the convex hull of B_2 ($CH(B_2)$) then there may be some intersection and as such, further checks are required. If however, this is not the case then (likely) the two curves do not intersect and the function can exit early.

   This is a simplified version of the logic presented by Yap in his *Micro Phase* of intersection detection. However, in order to cover all edge cases and make this a reliable rule to follow a lot more precomputation is required (See Elementary curves & curve coupling in [18]).

   As such, I ended up removing this feature as the additional work required was too high for this project.

2. I instead instituted a simpler heuristic using bounded boxes (See Section 2.4.1). Using the `Luxor` library[20], I was able to detect whether two bounded boxes intersected, if they did not, further checks can be skipped.

3. If two curve segments have already been checked for intersection, why bother checking them again?

   This question was solved by trading computation time for memory, by constructing a hash table linking curve pairs to a tuple containing the approximate values for t representing the point of intersection (if present) and a boolean, when checking two curve sections, if they already exist in the hash table the remaining computation can be skipped with the pre-computed result returned instead.

   In most cases a check is performed at least twice. Given two candidate solutions for two distinct agents B_1 and B_2, when evaluating the

fitness of B_1 it is checked for intersections with B_2 and equally when evaluating the fitness of B_2 B_2 is checked against B_1, thus buy storing the first result, the second check can be skipped.

In reality, many curve segments are found repeatedly inside populations and as such many more duplicate checks can be skipped through this method.

4. I implemented a recursion depth limit so as to introduce an upper bound in computation before the function returns false. Without this I found that often the checks would repeat until it was searching for intersections between infinitesimally small curve sections.

5. I was also able to reduce the runtime of this function via the use of threading. Upon a subdivision two more threaded tasks are spawned and the lazy disjunction of their results returned. The optimal case for this approach is that the intersection occurs early on in the curve as the task results must be fetched sequentially in the order that they were spawned.

My method keeps track of the divisions made and using this keeps track of the t value which maps P_n from the subcurve to the initial curve. The values of t are then returned if a intersection is found, this allows for the distances from the origin points to the point of intersection to be easily calculated using De Casteljau's algorithm along with my function for finding the length of a Bézier curve.

Upon each recursion, the diameter of the control points of both curves is calculated, if it is below a threshold ε, true is returned along with the estimated t values for each of the curves. Otherwise, unless any of the previously stated early termination criteria are met, the curve with the largest diameter is subdivided and two new threaded tasks are spawned with the lazy disjunction of their result returned.

An illustration of the recursive subdivision method described above can be seen in Figure 4.9

4.3 Macro-Level Planning

The next stage in development is to attempt to implement the wrapper to solve Problem 1 using the solution to Problem 2. I.e. to go from planning for multiple agents in a single section of road to planning for multiple agents across a network of interconnected roads.

4.3.1 Road Graph Construction

As specified in Problem 1, we require a description of the road space as a graph, $\mathcal{R} = (V, E)$ of intersections, V and road segments E.

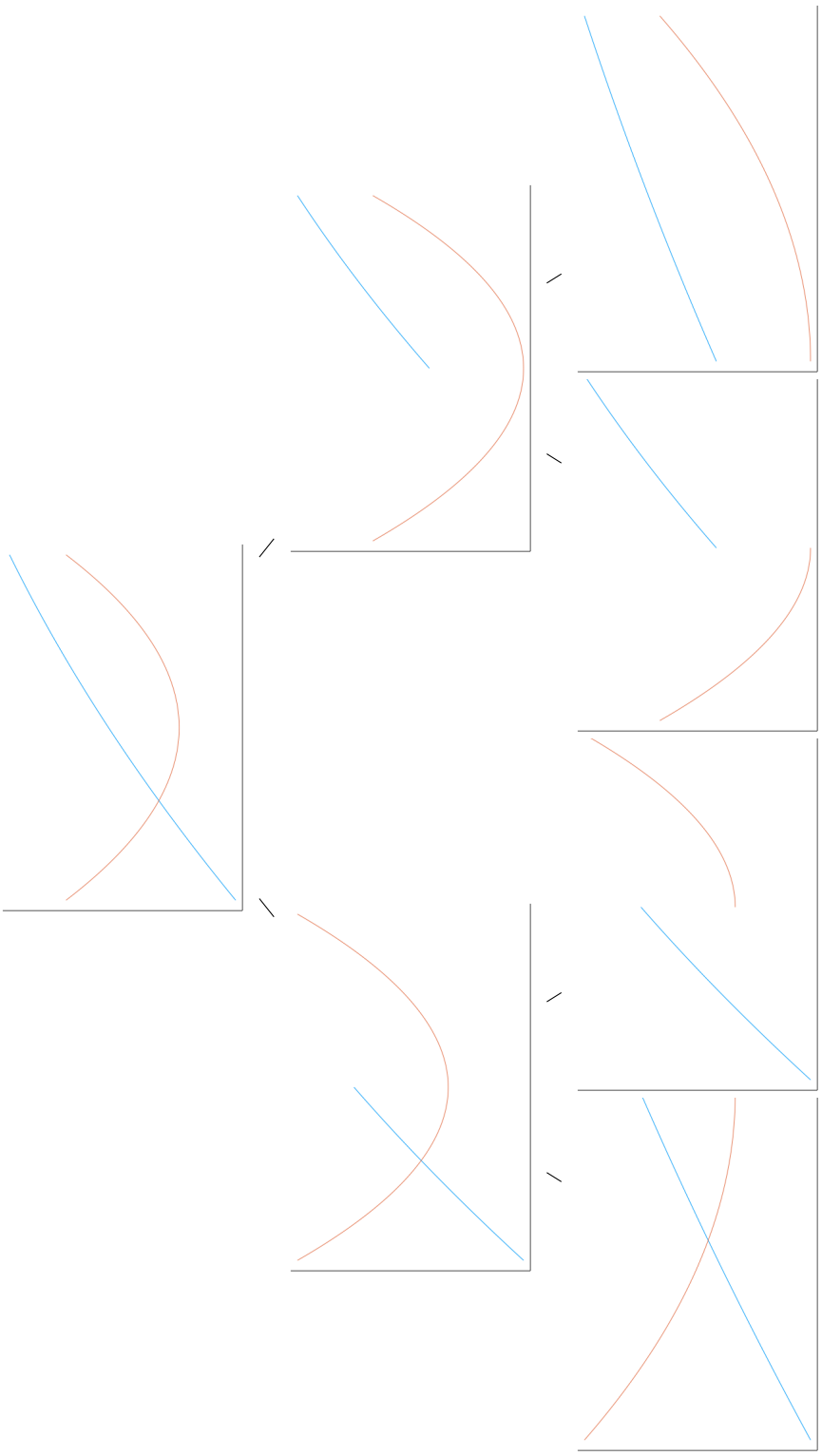


Figure 4.9: Illustration of Bezier Intersection recursive subdivision method (Recursion depth =2)

This is a malleable definition allowing for a lot of information to be encapsulated whilst maintaining an abstract canonical description compatible with a swathe of mathematics through graph theory.

I decided to use a mixture libraries to construct my road network: `Graphs.jl`[21], `LightGraphs.jl`[22] and `SimpleWeightedGraphs.jl`[23]. The graph structures found in `Graphs.jl` allow for a lot of data (The entirety of the `Road` structure can be stored as edge metadata) to be encapsulated but do not offer the same functionality as those found in `LightGraphs.jl`, `SimpleWeightedGraphs.jl` utilises the `LightGraphs.jl` format but allows for weighted edges when applying the various standard graph-based algorithms (including Dijkstra), as such I created my own helper functions to convert between the two formats, which is a relatively simple process of filtering data into a new, more restrictive, structure.

In the simplified weighted digraph format, the edges are weighted by their length. Going forward it would be possible to incorporate more advanced heuristics into this, mimicking the method by which applications such as Google Maps plan routes. This could include congestion, road surface quality, known planned road obstruction such as maintenance or a predicted large influx of vehicles, for example the exodus of people from a large sports arena.

A visualisation of this format can be seen in Figure 4.10 with edge names and their respective weights labelled.

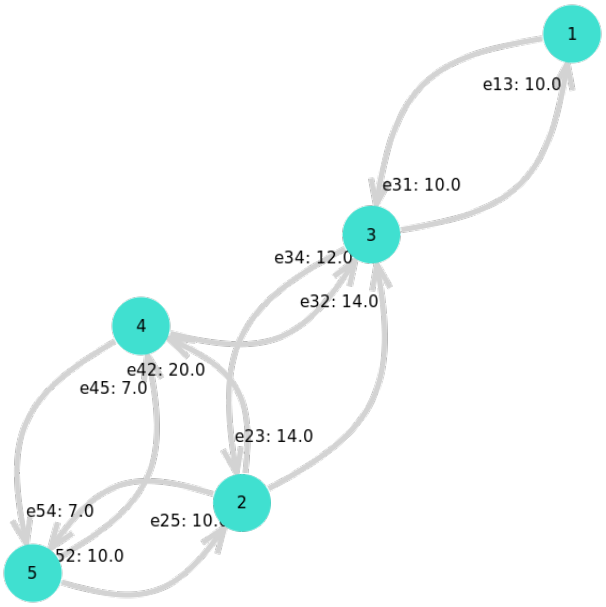


Figure 4.10: Example Road Graph, edge names take the form e<start><end>: <edgeweight>

With the graph constructed the next stage is to plan what I will refer to as the *macro* routes, i.e. the set of edges the agent must travel along $\{e_1, \ldots, e_m\} \subseteq E$ to reach its destination intersection, $d \in V$.

4.3.2 Macro-Route planning

Macro-route planning, as described in Problem 1, is the process of taking a set of intersection pairs corresponding to agents and producing the set of edges which connect the vertex pairs in the shortest distance. This task is almost the exact specification for Dijkstra's algorithm [24] for finding shortest paths in a connected digraph and as such this is the algorithm I employed to solve it.

The `LightGraphs.jl` library already implements this algorithm and the weighted element is incorporated by the `SimpleWeightedGraphs.jl` sub-library. As previously mentioned, the edges of my road-graph are weighted by their lengths.

For example the following construction of inputs for Problem 1 over the graph seen in Figure 4.10:

$$
\begin{aligned}
&A = \{a_1, a_2, a_3, a_4\} \\
&\mathcal{V} = [(1,4), (4,2), (1,4), (3,5)]
\end{aligned}
$$

Would yield the following abstract routes for the agents:

$$
\begin{aligned}
&P = \{ \\
&a_1 \mapsto \{1,3,2,5,4\}, \\
&a_2 \mapsto \{4,5,2\}, \\
&a_3 \mapsto \{1,3,2,5,4\}, \\
&a_4 \mapsto \{3,2,5\} \\
&\ \}
\end{aligned}
$$

The next stage in the process is to group these agents into *planning groups* i.e. groups in which agents will occupy the same road at the same time thus requiring cooperative planning.

4.3.3 Agent Grouping

When planning for multiple agents across multiple road spaces, routes may share the same stretch of road during route execution.

In the above example, it is trivial to see that agents a_1 and a_3 will share the same road at each stage of their route as they are following the same abstract path. We however, must also be able to determine whether agents

a_1 and a_4 share road $e25$, it occurs in both of their routes but one agent may have left that stretch of road before the other enters.

This problem is solved using my algorithm seen in Algorithm 2 . For instance, on the example input above, we would expect to see the following grouping:

$$
\begin{aligned}
&(3,2) \mapsto \{\{a_4, a_1, a_3\}\},\\
&(4,5) \mapsto \{\{a_2\}\},\\
&(2,5) \mapsto \{\{a_3, a_1\}, \{a_4\}\},\\
&(5,2) \mapsto \{\{a_2\}\},\\
&(1,3) \mapsto \{\{a_3, a_1\}\},\\
&(5,4) \mapsto \{\{a_3, a_1\}\}
\end{aligned}
$$

Where you can see a_4 occupies edge $e25$ at a time separate from agents a_1 and a_3.

Essentially, Algorithm 2 works as follows:

1. It initialises the return array, which is an array which, for each road in the graph, stores an array of sets of agents which occupy it concurrently at some point in time.

2. for each agent's macro-path as calculated using Dijkstra's algorithm, (see above for an example) construct a running total distance for the route. Call this agent i For the example $a_1 \mapsto \{1, 3, 2, 5, 4\}$ above, we would get:

$$[0, 10, 24, 34, 41]$$

   (a) for each edge in the macro-path, initialise an array of agents sharing the edge, initially only contains our currently considered agent.

      i. For each other agent's macro-path, call this agent j, check if it contains the currently considered edge, if it does, calculate the distance from the origin of j to the start of this edge. If this is less than the distance from the origin of i to the **end** of this edge, then they are said to occupy the same edge at the same time as i cannot feasibly leave the edge before j enters. In this case, append j to the array of agents sharing this edge with i, otherwise do nothing.

   (b) Upon completing this process for each agent, append the array of agents sharing this edge with i to the previously initialised array corresponding to this section of road.

3. Once all agents have been considered as i, return the constructed array of arrays of sets of agents sharing edges.

Algorithm 2: Path Grouping, `getPathGroupings`

Input:

- a road network graph $\mathcal{R} = (V, E)$
- a macro-path for each agent, P.

```
Initialise map of roads, roads
for each road, r ∈ E do
    roads[r] = [ ] ;          // Initialise list of agent groups
for each agent's macro-path, i ∈ P do
    Construct a running total of distance for each edge.
    for each edge, e ∈ E in the macro-path do
        ;    // Initialise routes sharing edge e with agent i
        microPlanAgents = [ i ]
        for each other agent's macro-path, o ∈ P\i do
            if e ∈ o then
                ;   // routes i and o share an edge (e) at some
                 point
                if length of i from origin to end of e > length of o
                 from origin to start of e then
                    ; // routes i and o occupy the edge e at the
                     same time
                    append(microPlanAgents, o)
                else
                    ;    // route i exits edge e before o enters
                    continue
        append(roads[e], microPlanAgents)
return roads;        // A map of edges to a list of lists of
 routes sharing that edge at the same time.
```

Once these grouping have been constructed, the next stage is to plan the low-level curves representing the paths each agent must take through each road segment, avoiding any agents it shares a given road segment with.

4.3.4 Multi-agent route generation

This process is a wrapper around the previously implemented parallel planner described in Section 4.2.

Informally the algorithm starts by generating the series of road edges an agent must navigate to reach it's goal intersection. This is done using Djikstra's algorithm.

My path groupings algorithm as described in Algorithm 2 is then applied to these *abstract plans*.

For each edge in each agent's path, if the edge has not already been planned, a set of routes for each agent shown to be occupying the edge concurrently with the current agent is generated using my parallel cooperative genetic algorithm as described in Section 4.2. When generating the start and goal points for each agent in a road, if a previous plan is known for the agent, the new start position will the same y coordinate as the previous goal position. Newly generated positions are calculated by evenly distributing points across the width of the road.

Once generated, these paths are appended to their corresponding agent's overall plan in the order in which they are executed. Once all plans have been made, this set of sets of routes is returned. A formal definition of this process can be seen in Algorithm 3

Algorithm 3: Macro-route planner

Input:

- set of start-goal pairs for each agent, $\mathcal{V}$
- a graph representing a road network, $\mathcal{R}$

$P = djikstraShortestPath(\mathcal{R}, \mathcal{V})$
pathGroupings = `getPathGroupings`$(\mathcal{R}, P)$
routes = [] ; `// initialise routes array as list of paths through each road segment specified by macro-route in` P
for *each macro-path,* $p \in P$ **do**
  for *each edge* $e \in p$ **do**
    concurrentAgentSets = pathGroupings[e]
    for *concurrentAgents* $\in$ *concurrentAgentSets* **do**
      construct start and goal positions for each agent,
      paths = parallelPlanner(startPositions, goalPositions, e) ;
      `// Parallel planner, solution to Problem 2`
      for *each agent* $a_i \in \mathcal{V}$ **do**
        append(routes[i], paths[i])
return *routes*

My approach vastly simplifies the problem of intersection navigation and linking the routes between road segments, essentially assuming all road segments are sections of a single long road which line up exactly. Papers such as "Automated Optimization of Intersections Using a Genetic Algorithm" by

Cruz-Piris et al.[12] begin to tackle this problem through the use of cellular automata (building on the work of Maerivoet and De Moor[25]) and GAs (See Section 3.1.1) but substantial work would be required to link our two systems together.

4.4 Language Choice

I have chosen to implement my approach using the Julia language project[26].

Julia is a relatively new language first developed in 2012 by Jeff Bezanson, Stefan Karpinski and Viral Shah. It is a multi-paradigm language allowing for functional, object oriented (OO) and meta programming approaches to problems. I will mainly be using it for it's functional and OO capabilities.

Julia operates using multiple dispatch similar to languages such as Haskell. It interoperates with C and Fortran codebases without the need of middle-man bloat. This fact allows it to utilise the extensive high performance C libraries for floating point operations. Julia is eagerly evaluated, uses a Just in time compiler and has a garbage collector.

Julia features a syntax similar to both Python and Matlab with performance on par with C. As I am already very familiar with python and have studied functional programming in a number of modules; I found this language very quick and intuitive to learn and the resulting code to be clean, idiomatic and fast.

A real-world deployment of a system based on my research would undoubtedly be required to run on small, relatively low performance, embedded systems and as such Julia may not be appropriate here. A language such as C or Rust may be used instead.

Julia also has distributed computation facilities. This sort of functionality could be extremely useful in a system such as mine as it could allow for computation to be spread across the agents themselves, removing the need for a central planning centre which could be a single point of failure.

5 Evaluation

This section of my report contains many 3 dimensional plots. I have included static captures and attempted to capture them at angles which demonstrate the points I am making. I have hosted all 3D plots seen below and they can be found indexed here: https://barrett370.github.io/Y4-Diss/posts/figures. I feel the data is better understood and expressed using these interactive plots. I have also included the raw html files in with this submission, see the "figures" directory.

5.1 Genetic Algorithms

The core of my approach to solving the problems outlined in Chapter 1 was creating a Genetic algorithm to evolve populations of candidate routes, returning the fittest once the termination criteria are met.

GAs are transparent but stochastic approaches to optimisation problems. As such, it is much easier to examine the operations and decisions they make compared with *black box* approaches such as neural networks, however due to their random nature, it can be difficult to predict their exact behaviour and results can differ greatly from run to run if the number of generations or population size is too low.

Whilst a poor choice of training data can lead to unintended/ unpredictable behaviour from approaches like neural networks, they have the advantage that most of their compute time is used when constructing the initial model, subsequent usage of these models requires little compute time. Whereas, GAs must run the entirety of the *learning* process from scratch whenever a new route or set of routes is required. In an application where the system sees a high number of unique requests, this compute time can quickly amount to much longer than the training time for a neural network.

5.1.1 Genetic Operator Performance

During development I implemented two different operators in each category, but there are many other approaches proposed in academic literature. Given more time, I would have liked to implement more and present a more empirical evaluation of each.

A complete table of runtime results for each procedure in my PCCGA can be found in Table A.1

Selection

In a single generation, over three populations of 15 individuals, my ranked selection operator ran 3 times totalling $75.1\mu s$ or an average of $25\mu s$ per run, a negligible amount of time relative to the overall runtime.

As you can see in Figure 4.5, my ranked selection operator performs its task well, removing the obviously less fit individuals in favour of more copies of fitter ones. This can be seen by the removal of the line that dips below the x axis (in this example the bottom road boundary was defined as $b_1(x) = 0$) and the fact that duplicate routes are found as indicated by the 3rd graph of unique routes showing fewer individuals than the second.

Ranked selection compared with fitness proportional (roulette wheel selection) is much more predictable, it may not explore as much of the search space but it is more likely to thoroughly explore a single area. Most other selection operators seem to focus on maintaining a diverse population so as to not get stuck in local minima.

Mutation

The mutation operators I implemented were Uniform and Gaussian.

I found Gaussian to perform better in cases where a route may be close to optimal in the phenotypic space, i.e. with minimal changes to trajectory, it would be optimal. In such cases the genotypic fitness may not accurately represent this, a route that collides with another or passes through infeasible space will suffer from a large genotypic fitness penalty possibly encouraging it to be removed from the *gene pool*. Gaussian mutation, preferring mutated points closer to their initial position can be effective as moving such routes closer to the optima.

However, in cases where the initial population generation has created all routes far from the optimal, Gaussian mutation can struggle to generate mutated points extreme enough to direct the GA towards the minima.

Part of the problem with Gaussian mutation, and mutation in general on n-degree Bezier curves, is that even a relatively major control point movement may cause only minor changes to the overall trajectory of the curve in cases where n is high.

An example of a relatively large control point mutation leading to a minor direction change can be seen in Figure 4.6.

It is possible that an approach incorporating a form of Simulated Annealing could help to broadly explore the search space in early generations before refining solutions in a found minima in latter generations. This could operate by applying a different mutation operator such as Uniform mutation in the first 60% of generations before swapping to Gaussian when we would expect our algorithms to have found some local minima.

In a single generation, over the same 3 sets of 15 individuals, my Gaussian

mutation operator ran 3 times totalling $1.71ms$ runtime, an average of $571\mu s$. This is an insignificant amount of time when compared to the overall runtime of this task which stands at 14.4.

Crossover

I implemented both single point and k point crossover. I ran my benchmarks using single point crossover in an attempt to minimise the amount of randomness and differences in convergence speed between samples.

Simple crossover amounted for a insignificant proportion of the runtime of 3 sets of 15 agents over a single generation, totalling 0.01% of the runtime at 1.14ms, averaging $381\mu s$ on each of its 3 runs.

k point crossover has little additional runtime complexity, essentially applying the simple crossover operator k times, although in practice the runtime isn't this simple due to the random nature of the k value.

Fitness

As previously mentioned the most important operator, and the one I spend the most time working on, is the fitness function.

This acts as the objective function for the algorithm as it seeks to minimise its value across multiple agents over multiple generations.

This operation, especially when modified to detect collisions, was the source of most of the runtime of my project. In the toy example of 3 sets of 15 agents across a single generation, the fitness function accounted for a large proportion of the runtime at 14.4 across 3 calls, an average of 4.81 seconds per invocation. This is after all my attempts to speedup the process (See Section 4.2.1).

The final incarnation of my fitness function can be thought of as two separate parts:

1. The *base fitness* i.e. the fitness when considered in isolation in the road space.

   This calculation again can be thought of as being comprised of 3 parts as outlined in Equation 4.1:

   In total this portion of the function ran 135 times in an average of $55.9ms$ for a total of 52.3% of the fitness function runtime

   (a) Route length
   This procedure had an average runtime of around $5.36\mu s$, running 135 times it amounted for 0.01% of the fitness function runtime.

   (b) Infeasible distance length taking an average of $32.7ms$ to run over its 135 runs, making up 30.6 of the fitness function runtime.

(c) Close proximity distance length took an average of $23.2ms$ over 135 runs totalling 21.7% of the runtime

2. The collision penalty

   Collision detection as previously stated was a major source for complexity in my project. However, after a lot of tweaking and time saving measures, in this toy example the runtime for the collision detection stands at an average of 24.8ms, around the same as the runtime of the high proximity distance calculation.

5.2 Bézier Curves

Bézier curves have been utilised in this project to encode and represent the route of a vehicle. As mentioned in Section 2.2, there are many reasons I originally selected them for this task. However, over the course of implementation and testing, a number of downsides have been presented.

One such downside is that objective numerical approaches with Bézier curves are often complicated, expensive and do not generalise well to n control points.

This leads to many heuristics, and approximations being employed to save computation. Approximations and assumptions in a system as the one proposed in this report, are sub-optimal and could potentially lead to undesired behaviour, which could ultimately have dire consequences if such a system were to be deployed.

Other research such as that by Cai & Peng[9] takes a different approach, using discrete, grid-based search spaces in which routes are made up of a series of connected straight line segments. This approach removes much of the complexity from my solution but introduces its own concerns.

The routes generated by Cai & Peng's approach are intended to be executed by autonomous robots, so no thought has been given to potential passengers. Consequently, these routes would require smoothing as a post-planning process, re-introducing complexity.

Another possible representation is the approach taken by Cruz-Piris et al.[12] which involved representing the section of road, in their case an intersection, as a cellular automata in which a single vehicle can occupy a single cell at any given point in time.

There were however, also some advantages and nice properties of Bézier curves which lent themselves to the task.

Their relatively simple abstract construction as a series of control points proved easy to represent concretely as a genotype. This made the creation of the various genetic operators relatively simple, requiring little pre or post-processing.

They are also capable of representing a high complexity of curve in a relatively simple and concise manner. This makes code much more approachable and algorithms easier to digest.

5.3 Single Agent Planning

The majority of the components of my single agent planning system have been evaluated in Sections 5.1 and 5.2. Here I will discuss its overall effectiveness and any possible extensions that could be implemented given more time.

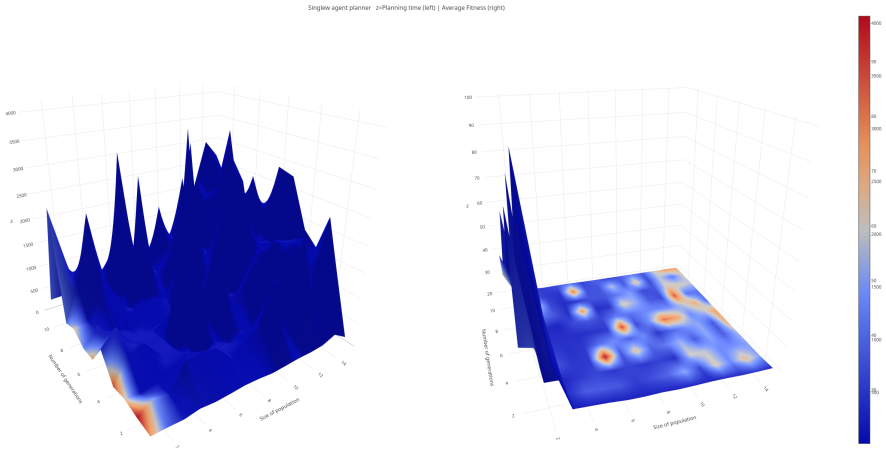


Figure 5.1: Single agent planning: number of generations against size of population against planning time (left), fitness (right), over *easy* road space (Figure 5.5a)

In general, I feel that my single agent planner performs well[1]. In a road space such as the one shown in Figure 5.5a my algorithm converges to within 10% of the fitness of a direct line between the start and goal point within 2 generations over 4 individuals as seen in Figure 5.1.

As the road space complexity increases (See Figures 5.5b,5.5c), the speed at which my algorithm convergence is reduced.

Over the road space seen in Figure 5.5b, this is only by a small margin, taking closer to 3 generations (See Figure 5.2).

However, on a more difficult example such as that seen in Figure 5.5c, all routes close to a straight line are infeasible, resulting in a lower average fitness as can be seen in Figure 5.3, as well as a higher average planning time as it is required to perform more involved infeasible distance calculations more often.

[1]**Note:** all data used in creating the figures referenced in this section can be found in the *sa-all-data.csv* file included with this report

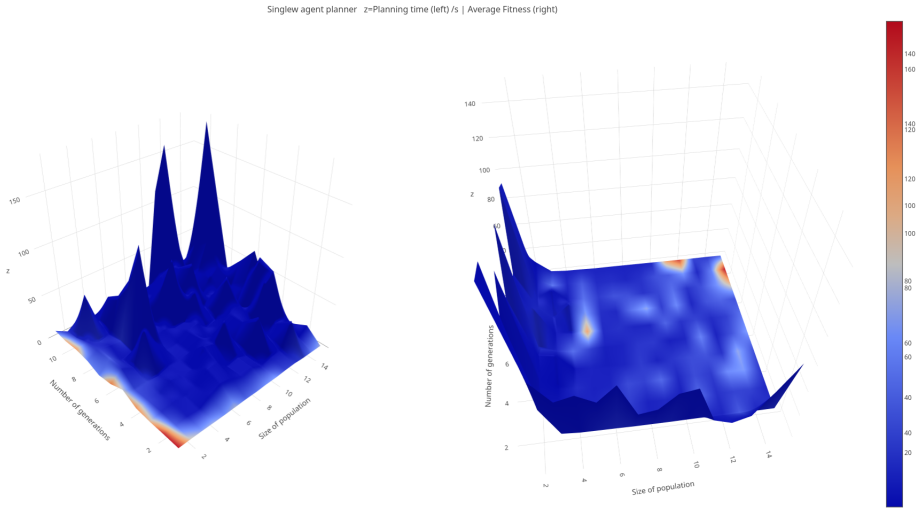


Figure 5.2: Single agent planning: number of generations against size of population against planning time (left) /s, fitness (right), over *moderately difficult* road space (Figure 5.5b)

On the example road shown in Figure 5.5d, the algorithm converges slower still, but still reaching what I would consider to be a *good* solution within 5 generations with a population size around 7, these results can be seen in Figure 5.4. In terms of planning time, with a few anomalous exceptions planning time steadily increases roughly proportionally to (the number of generations $\times$ the population size).

In general, the analysis of the planning time for my algorithm is difficult due to the inherent stochastic nature of Genetic algorithms. As such, you may notice many erroneous spikes in seemingly random places across the results surfaces, however, in all cases a general upward trend in planning time is seen.

In Figure 5.6, you can see the relationship between solution complexity (quantified by the average number of control points in the fittest solution), number of generations, size of population and difficulty of road section. Generally, in all subfigures, you can see a trend to lower solution complexity as the number of generations increases. This would make sense as the GA has had more opportunity to explore the search space, settling on a more optimal solution. This could be potentially aided by incorporating a measure of solution complexity as a penalised component of the fitness function. Whether this would produce intended behaviour warrants further research.

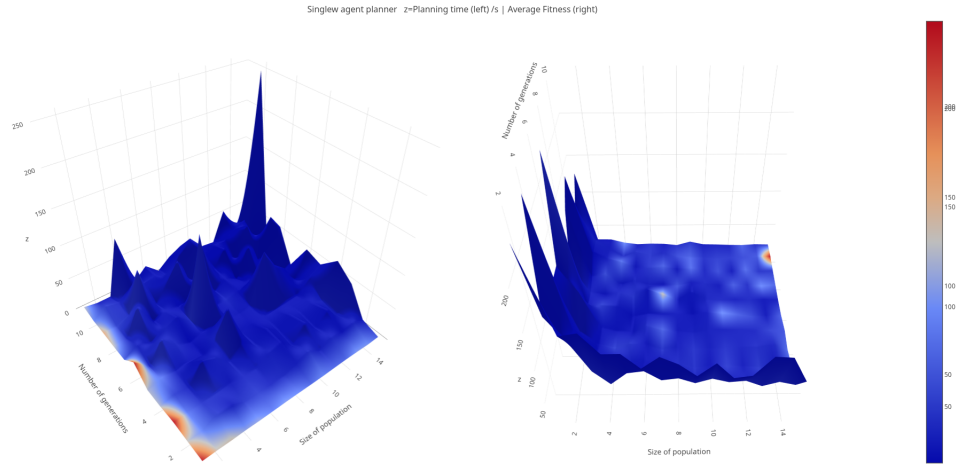


Figure 5.3: Single agent planning: number of generations against size of population against planning time (left) /s, fitness (right), over *difficult* road space (Figure 5.5c)

5.3.1 Extensions

The tuning of parameters is something that I feel requires more time and would lead to more consistent and accurate results. An additional extension that would make the generated routes more feasible is to add considerations to the routes velocity profile and turn angles into the fitness function. Generating Bézier curves with these considerations has been done in other research such as that by Chen et al.[27] in 2014 in which they generate routes whilst considering the route curvature and velocity constraints, fixing the degree of Bézier curves used may have made this task easier in their case.

5.3.2 Comparison with other work

The closest work to my single agent planner is that outlined by Kala in the 6^{th} chapter of his book "On-Road Intelligent Vehicles" in which he details a GA approach using Bézier curves to route a single vehicle through a section of road. It is difficult to directly compare our solutions as Kala does not provide a codified solution and his evaluation does not go into great detail, infact the units are missing from the few graphs he does provide.

Kala does incorporate velocity profiles into his GA, this is an area I have already outline something I would have liked to have added given more time.

Kala's approach seems to rely on much larger populations in order to generate feasible routes, showing a system using Cartesian coordinates (like mine) to require a minimum of 40 individuals to plan a route which did not leave the road-space, as can be seen in Figure 5.1, my solution requires fewer

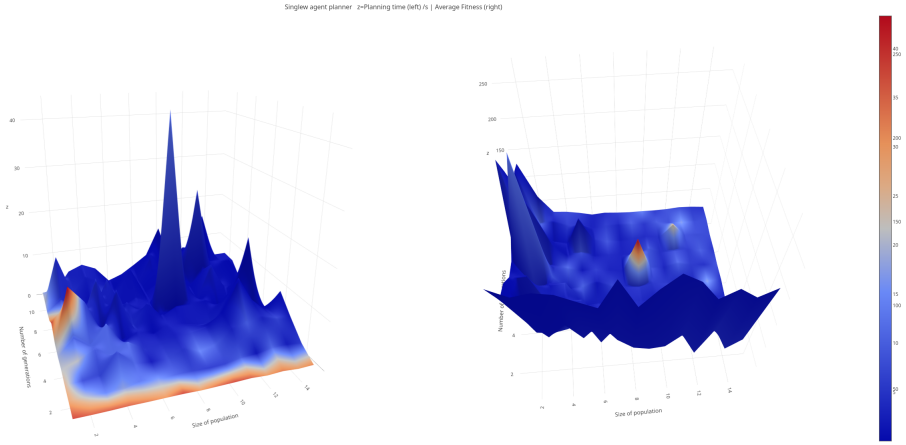


Figure 5.4: Single agent planning: number of generations against size of population against planning time (left) /s, fitness (right), over *difficult* road space (Figure 5.5d)

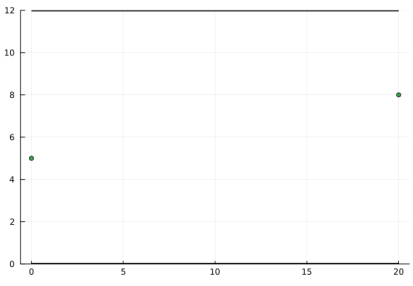
than 10.

The approach seen in the paper from Elshamli et al.[10] also did not provide concrete results and as such I struggle to directly compare our approaches. A key factor in their path planning was the path *smoothness*, their routes were made up of straight sections between nodes, not unlike the control points used to form a Bézier curve. Their *smoothness* metric evaluated the angle between the route segments joined by a node, if the angle was too great, it was deemed infeasible and penalised. By virtue of using Bézier curves I feel I have intrinsically included this requirement as by their nature Bézier curves are smooth and continuous, further requirements as to the change in y coordinates could be imposed to ensure turns are not too *tight* .

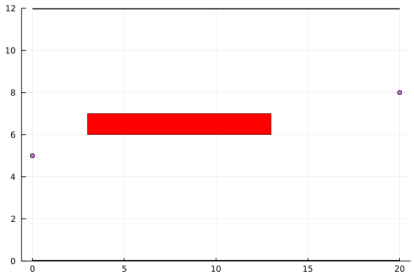
5.4 Cooperative (Multi-agent) Planning

[2]My solution to the problem of planning n non-colliding routes for n agents was to wrap my existing `GA` function in a cooperative *layer*. This cooperative layer relied on a function for detecting collisions which had extremely high overhead, at one point causing around 50x slowdown in the running time of the function. Detecting intersections between two Bézier curves is a non-trivial task with the best methods taking the same approach of recursive subdivision that I utilised.

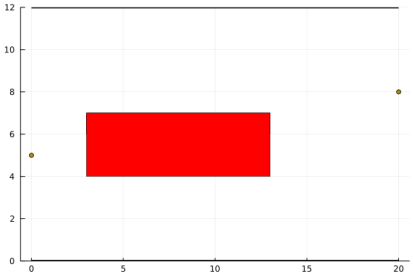
[2]All data used in constructing the figures seen in this section can be found in the file 'ma-all-data.csv' included with this report



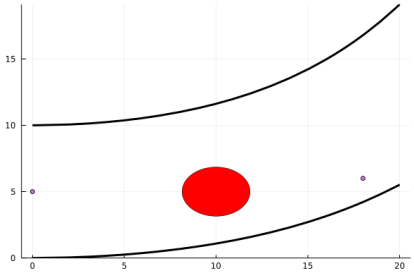
(a) Easiest road segment, no obstacles, straight



(b) Second easiest road segment, single, small obstacle in a straight road. Approx. 4% of road is infeasible

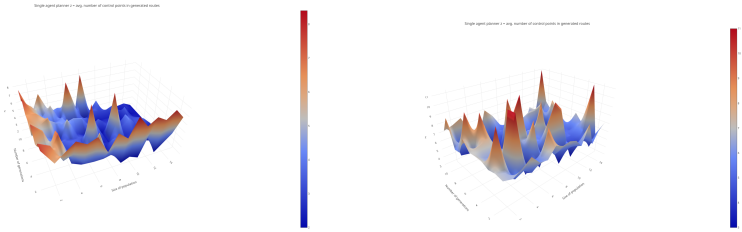


(c) Road segment with single, large obstacle in a straight road. Approx. 12.5% of road is infeasible.



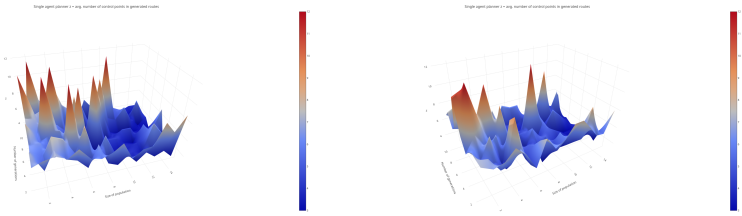
(d) Road segment with single, medium sized obstacle in a curved road. Approx. 4.9% of road is infeasible.

Figure 5.5: Roads used to test single agent planner performance, start and goal positions shown, ordered in terms of difficulty



(a) Average number of control points in routes for Road 5.5a

(b) Average number of control points in routes for Road 5.5b



(c) Average number of control points in routes for Road 5.5c

(d) Average number of control points in routes for Road 5.5d

Figure 5.6: Average solution complexity for a single agent over a varying number of generations and sizes of populations across various roads

As detailed in Section 4.2, I attempted to speedup my cooperative planning layer through many different means, including parallelising the entire method as well as individual high-complexity sections such as the Bézier curve intersection function. The system on which I have been benchmarking performance is equipped with 16 cores running at a maximum of 4.2GHz.

Julia also boasts the ability to distribute work across multiple systems over secure shell, this could conceivably allow for on-board vehicle computers to aid in the computation and planning of their own routes. Each individual in a concurrent plan is spawned as a unique threaded task and as such having cores equal to the number of concurrently planned agents is ideal. Julia also has good support for GPU programming through libraries such as `CUDA.jl`[28], in their 2018 paper Roberge et al.[13] showed how effective massively parallelising genetic algorithms can be, seeing a 290x speedup when compared to sequential execution on a CPU. These two factors make the relatively high planning times seen in this report potentially immaterial and could allow for much higher numbers of generations or larger populations to be feasible.

I tested my cooperative planning method against the same road sections I used to test the single agent planner and unless stated otherwise tested with 3 agents with start-goal positions seen in Figure 5.14. In all cases I used Gaussian mutation, ranked selection and simple crossover.

When considering the easiest road section, Figure 5.14a, the average fit-

ness of individuals at low generations and population size is incredibly high, leading to me thresholding it at 40 when producing Figure 5.7. However, it quickly drops producing *good* solutions with average fitness in the region of 18 to 20 within 4 generations over populations around 5 in size.

Planning time sees typical steady growth as the number of generations and population size increases, peaking at around 1 minute (excluding anomalous results) but averaging closer to 40 seconds at the higher end of population size and generations.

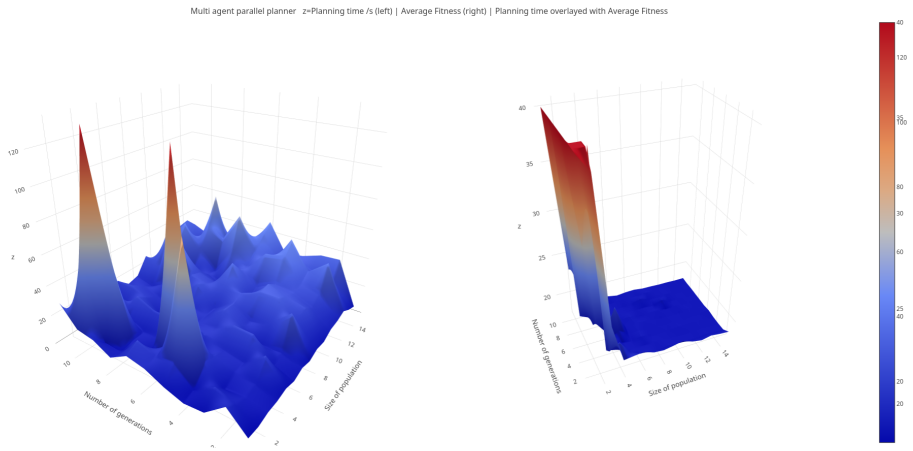


Figure 5.7: Multi agent planning number of generations against size of population against planning time (left), fitness limited at 40 (right), planning through road seen in Figure 5.14a

On a road with an obstacle (Figure 5.14b), the result seen in Figure 5.8 is achieved. Taking slightly longer to converge to *good* solutions at around 6 generations over 6 population. It appears that the number of generations has more of an effect on fitness with 6 generations over 5 populations showing better results than 4 generations on 7 population. However planning time appears to suffer more from more generations than it does from larger populations.

Increasing the difficulty further, over the road seen in Figure 5.14c, we see a yet slower convergence rate with consistently *good* results appearing after 8 generations over 8 individuals, we do however see the fitness drop steeply between the 7th and 8th generations, much more steeply than in the *easier* road segments. Planning time is significantly higher than in previous tests with an average planning rising to around 3 minutes on high numbers of generations and sizes of populations.

When planning on the most difficult road segment (Figure 5.14d), my approach appears to be much less stable. In Figure 5.12, you can clearly see the fitness still peaks above 200 regularly even after a high number of

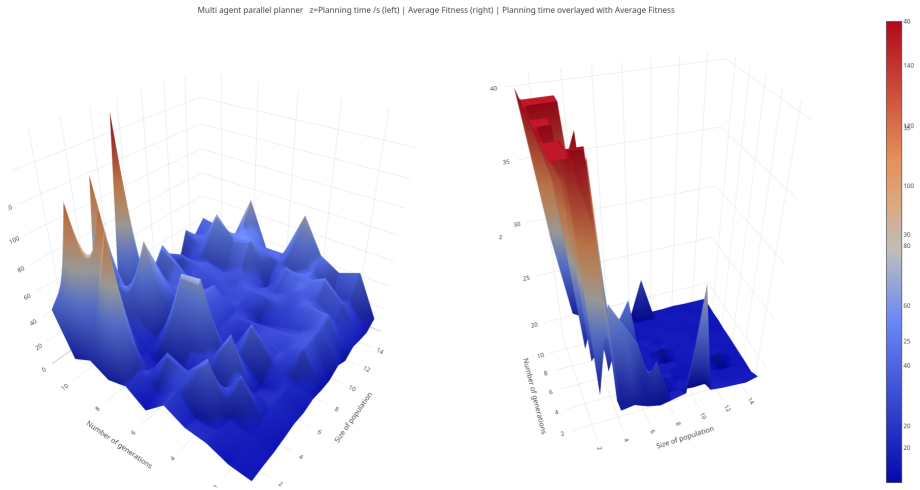


Figure 5.8: Multi agent planning number of generations against size of population against planning time (left), fitness limited at 40 (right), planning through road seen in Figure 5.14b

generation and/or population size. Whilst a general trend lower fitness can be seen, the regular spikes mean collisions or intersections with the obstacle are common, producing infeasible routes. Clearly more work is required for my approach to handle these more complex road spaces well, as well as perhaps higher populations / numbers of generations.

Planning time again is seen to trend upwards proportionally with (number of generations $\times$ size of populations), reaching very high planning times peaking at 280 seconds for 10 generations over populations of size 15 although, interestingly this is lower than the times seen when planning for Figure 5.14c, this may be due to the start and goal positions changing leading to more routes in which collision detection can be exited early, i.e. routes in which intersections occur early in the trajectory. This could also be due to external factors such as increased background load on my system during the planning of Road 5.14c

When varying the number of agents being concurrently planned for we can see that the planning time steadily increases with each new agent added. In Figure 5.11 where the planning time surfaces of runs using 1,2,3 and 4 different agents through the road seen in Figure 5.5a, the surfaces are shown to layer in order of number of agents, this can be more easily seen in Figure 5.10, in which the average planning time across all generation-population size variations is averaged for each number of concurrently planned agents in the range 1 to 4.

Figure 5.10 shows a trend very close to directly proportional, especially when one considers the amount of extra factors which could be affecting the

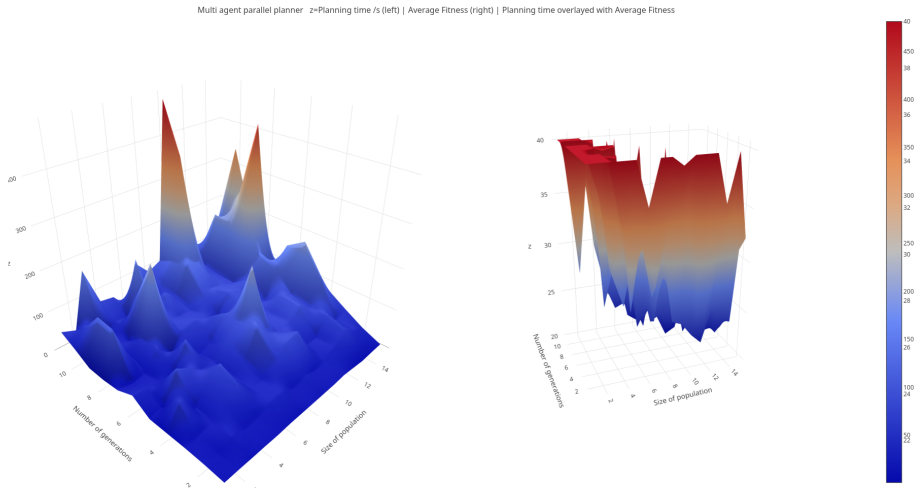


Figure 5.9: Multi agent planning number of generations against size of population against planning time (left), fitness limited at 40 (right), planning through road seen in Figure 5.14c

planning time (system load, randomly generated initial populations, random chances for operator to apply and random extents to which they apply). Each data point in Figure 5.10 is the average of 2250 results, this may suggest that at such high sample rates, the (pseudo) random nature of GAs can be ignored. The trend seems to follow the line $y = 11.2x - 8.6$, implying an additional 11.2 seconds of computation for each added agent. It is difficult to verify this as there are so many subcomponents with varying complexities tied to the number of agents, however, this linear trend is promising as it suggests steady growth in runtime as agents are added rather than a less manageable exponential or even **NP** time complexity.

As previously mentioned one of the key improvements I made to my initial approach to reduce planning time was re-writing it to take advantage of multiple cores, before this I wrote it to run asynchronously, taking full advantage of the single thread on which it ran. In Table 5.1 you can see a marked difference in average runtime over these different approaches. The average runtime of my final, parallel cooperative coevolutionary genetic algorithm (PCCGA), approach being around 2.6 times faster, when run over the 16-threads of the same CPU with the same 3 agent, single generation, 15 population toy example evaluated earlier.

5.4.1 Extensions

The extensions to this system are much the same as those outlined for the single agent planner: parameter tuning and velocity profiles.

Procedure	ncalls	time	avg
Sequential single threaded CCGA	3	97.2s	32.4s
Async (1 thread) PCCGA	3	86.1s	28.7s
PCCGA	3	37.6s	12.5s

Table 5.1: Comparative runtimes for various incarnations of cooperative planner

In addition to this there are certain new parameters and values introduced by the cooperative wrapper; the objective size of an agent is something which I have not explicitly fixed, by fixing this value the parameters associated to Beizer curve intersection detection and collision detection could be further refined to give the most accurate result, while minimising runtime.

As well as these improvements, there is the possibility outlined earlier of distributing and/or further parallelising this approach. We have seen a substantial reduction in runtime when increasing the number of available cores by 16, if run over the up to 10752 cores seen on modern enterprise GPUs such as the RTX A6000, even despite the much lower core clocks, we could see an incredible further reduction in runtime. Adding to that the distributed possibilities, the runtime of my current system could potentially become negligible, allowing for more dynamic real-time planning which is able to quickly adapt to unforeseen situations during route execution.

5.4.2 Comparisons with other work

My approach in its current form is much more abstracted from a real-world system than other research such as that by Kala, as mentioned previously, his proposed system incorporated a velocity profile as well as explicit vehicle sizes. His approach to multiple agents also hinged on inter-vehicle communication to plan overtakes as opposed to my approach using a plan-then-execute methodology, with all coordination taking place within the planner and each agent being assigned a completed route once planning has concluded.

Other research such as the work by both Cai & Peng as well as Crus-Piris et al. took a discrete approach to the search space, both using grids but the latter also utilising cellular automata, this allowed them to avoid the complex calculations and approximations I had to employ when assessing collisions or intersections with infeasible space. I feel that by viewing the road as a discrete search space, you create a need for a lot more post-processing on the routes in order for them to be smooth, possibly invalidating the guarantees given by the planner. By planning smooth, continuous routes directly, I can avoid these issues.

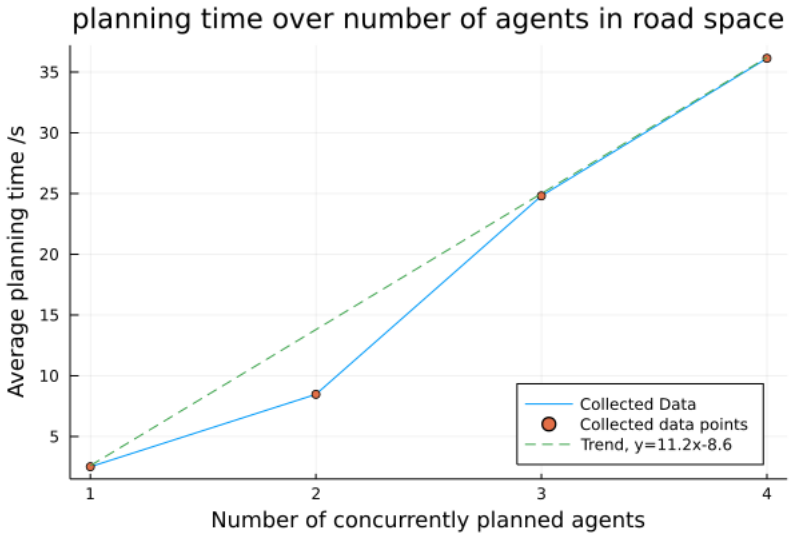


Figure 5.10: Varying number of agents being concurrently planned through Road 5.14a

5.5 Macro-level Route planning

The core of my macro-level planner was the previously evaluated multi-agent system, and as such I will not analyse its running time as the extra operations surrounding the invocations of the cooperative GA contribute relatively little time complexity.

My approach to routing multiple agents through a road network heavily relies on a number of naive assumptions:

- The constant uniform velocity profile of all agents

  I have spoken about this assumption when evaluating the multi-agent planner but this wrapper also relies heavily upon it.

  When assessing as to whether agents occupy a road space at the same time it assumes each agent takes the most direct route through any previous road spaces at the same speed at which all other agents navigate their previous road spaces. It therefore reasons that if the length of an agent's previous road space segments is less than another agent's previous distance + the distance of the road in question, they will share it.

  In reality, this question would be much more complicated, it relies on the knowledge of the actual path length of all agents as well as knowledge of their velocity profiles as they execute these routes.

  An extension to take this into account would require an almost complete re-imagining of Algorithm 2, unless a pessimistic view of route

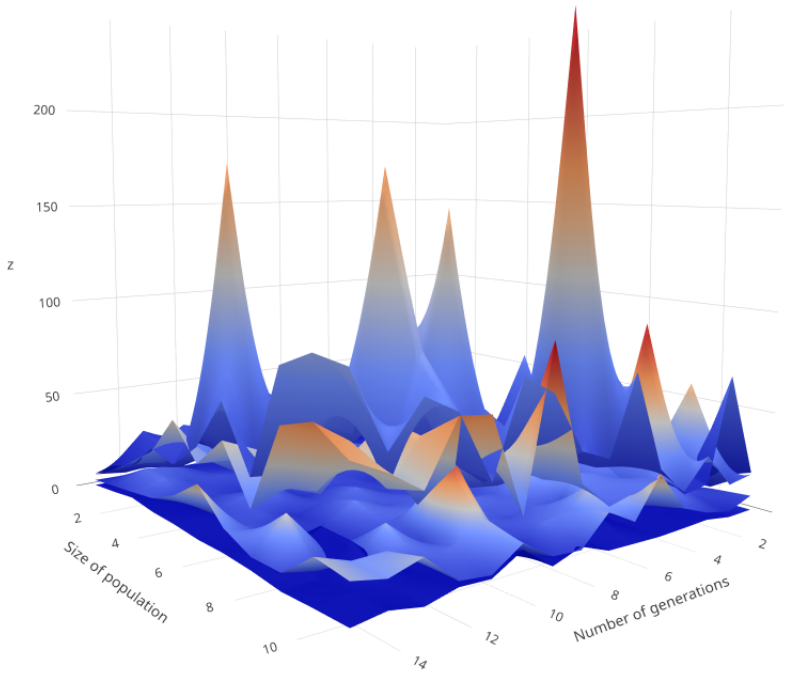


Figure 5.11: Overlaid surfaces for planning time of 1,2,3 and 4 agents through Road 5.14a over 10 generations and 1 to 15 population, z= planning time /s

efficiency is maintained.

- If two agents are grouped together, sharing a road section, their routes are planned under the assumption that they begin at the start of the road at the same time. Even under the assumption detailed above, this is not accurate. For the system to be extended to allow for agents to start at different points significant changes to the multi-agent planner would be necessary as collisions would need to be ignored until such a time as one agent has *caught up* to another, something that again relies on the implementation of generated variable velocity profiles.

- The connections between each road segment are also naively assumed to be non-existent, with one section of road flowing directly into another. Obviously, in the real world, there are a variety of types of junctions linking two stretches of road. However, designing a system which can route traffic through such junctions is a domain unto itself as seen in papers such that by Cruz-Piris et al.[12] in 2019.

- The assignment of goal points in the road space is also not ideal, simply splitting the width of the road between the concurrent agents and

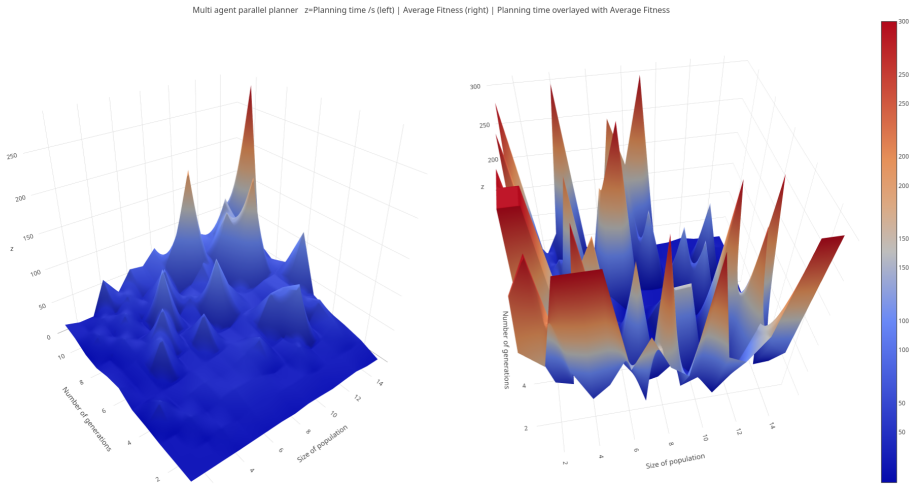


Figure 5.12: Multi agent planning number of generations against size of population against planning time (left), fitness limited at 300 (right), planning through road seen in Figure 5.14d

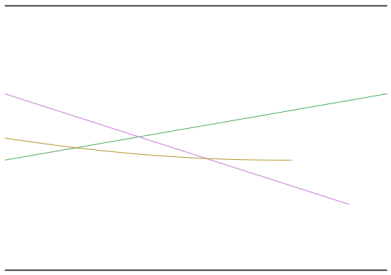
spacing them out equally, not taking into account initial start positions which leads to circumstances where an agent must cross the entire road, increasing the complexity of the planning problem an example of this can be seen in Figure 5.16a.

In general, I feel this section of my project was able to demonstrate how my underlying system might perform in a larger-scale environment, separate from the toy examples shown above. It performed well with only 3% of the overall runtime stemming from the macro-level wrapper, i.e. 97% of the macro planning time comes from the invocations of the parallel cooperative planner evaluated in Section 5.4. It is important to note that the performance of Dijkstra's algorithm is primarily related to the total number of vertices in the graph and as such the more intersections a road network has, the more expensive this wrapper will be, however the sparsity of the graph does also play a role.

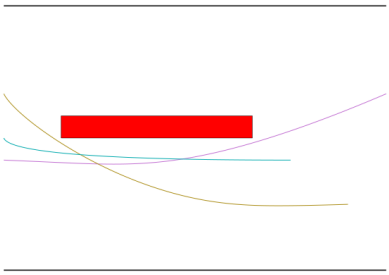
5.6 Codebase Evaluation

In total my codebase consists of around 2000 lines of Julia, a full breakdown can be seen in Figure 5.18

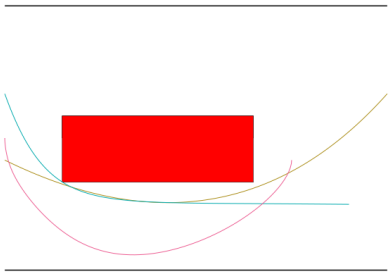
The areas which required the most work were the Bézier intersection procedure, parallelising the cooperative planner, designing the path grouping procedure for the macro level planner and creating the multitude of utility functions for plotting and otherwise visualising the results of the GA(s).



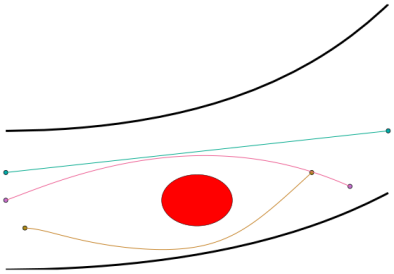
(a) Example routes through easy road space, average fitness = 18.0



(b) Example routes through medium road space, average fitness = 18.4



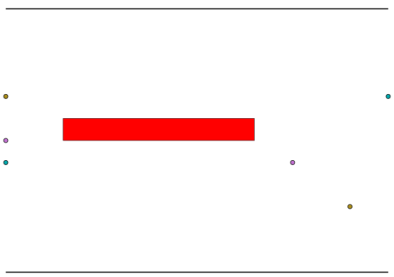
(c) Example routes through hard road space, average fitness = 155.8



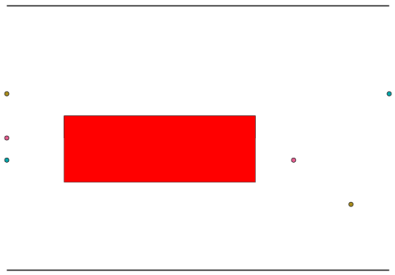
(d) Example routes through hard, curved road space, average fitness = 19.3

Figure 5.13: Roads used to test single agent planner performance, start and goal positions shown, ordered in terms of difficulty

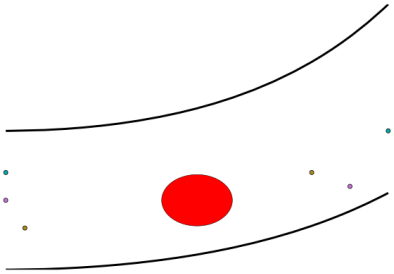
(a) Easiest road segment, no obstacles, straight



(b) Second easiest road segment, single, small obstacle in a straight road. Approx. 4% of road is infeasible

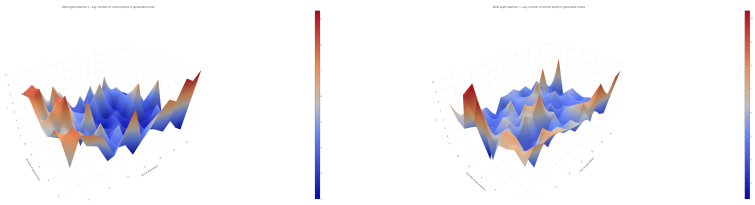


(c) Road segment with single, large obstacle in a straight road. Approx. 12.5% of road is infeasible.



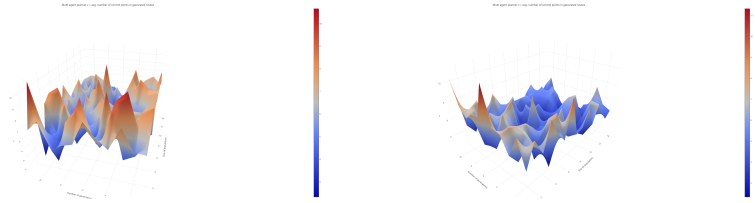
(d) Road segment with single, medium sized obstacle in a curved road. Approx. 4.9% of road is infeasible.

Figure 5.14: Roads used to test single agent planner performance, start and goal positions shown, ordered in terms of difficulty



(a) Average number of control points in routes for Road 5.14a

(b) Average number of control points in routes for Road 5.14b



(c) Average number of control points in routes for Road 5.14c

(d) Average number of control points in routes for Road 5.14d

Figure 5.15: Average solution complexity for a 3 agents over a varying number of generations and sizes of populations across various roads (See Figure 5.14)

The main body of my GA resembles the shape of the generic genetic algorithm seen in Algorithm 1, and can be seen in Figure 5.17.

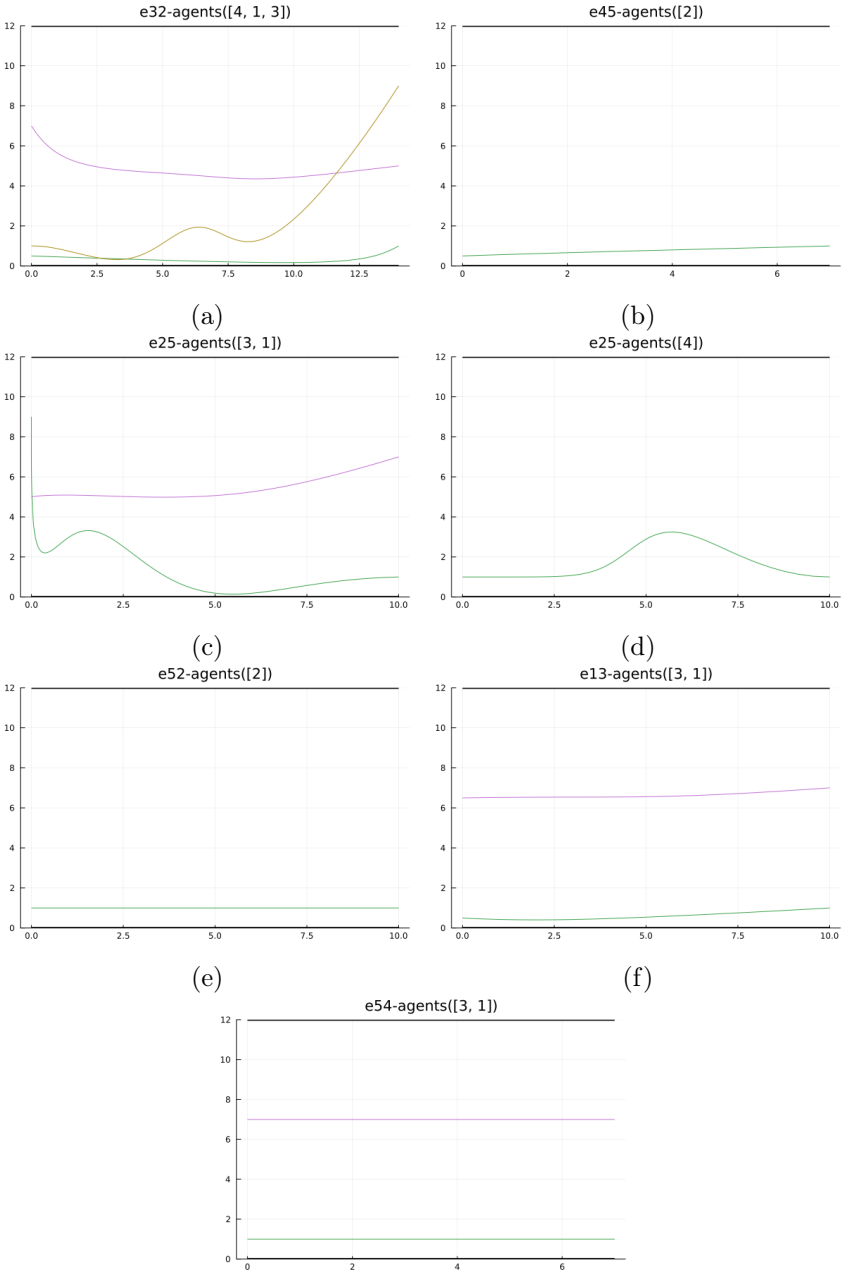


Figure 5.16: Example routes planned using macro route planner, example can be seen in detail in Section 4.3.2

```julia
P = generatePopulation(n, start, goal, road) #Generate initial population
map(p → p.fitness = p ▷ ℱ, P) # Calculate fitness for initial population
while n_gens > 0 && length(P) > 0
    P = (P
        ▷ P → selection(P, method=selection_method)  # Selection operator
        ▷ simple_crossover ▷ new_pop → append!(P, new_pop)  ## Crossover + append new individuals
        ▷ uniform_mutation! # apply mutation operator
        ▷ P → begin map(p → p.fitness = p ▷ ℱ, P); P end # recalculate fitness after mutation
        ▷ P → map(repair, P)  # attempt repair of invalid solutions
        ▷ P → sort(P, by=p → p.fitness) # Sort my fitness
        ▷ P → filter(isValid, P) # remove invalid solutions
        ▷ P → P[1:minimum([n,length(P)])]# take top n
    )
    n_gens -= 1
end
P
```

Figure 5.17: Core of my GA, written in Julia
(Note: x ▷ f ⇔ f(x))

Language	Files	Lines	Code	Comments	Blanks
C	2	61	48	4	9
Julia	19	2619	2077	235	307
Markdown	2	16	0	12	4
Python	1	76	58	6	12
Plain Text	1	2638	0	2638	0
TOML	2	1581	1300	1	280
HTML	31	2371	2188	6	177
├ CSS	1	14308	11777	925	1606
├ JavaScript	31	684	643	5	36
(Total)		17363	14608	936	1819
Jupyter Notebooks	18	0	0	0	0
├ Julia	14	885	795	16	74
├ Markdown	5	258	53	133	72
├ Python	1	45	44	0	1
(Total)		1188	892	149	147
Total	76	9362	5671	2902	789

Figure 5.18: Codebase statistics, provided by tokei

6 Conclusion

This is clearly a very broad and complex area of research. I have shown that the task of safely routing many vehicles through interconnected sections of road can be thought of as multiple, large, sub-problems, many of which are seeing highly targeted research.

I have shown the merits of my approach: the generative nature of GAs as well as their potential for massive parallelisation along with the smooth, continuous and malleable properties of Bézier curves have been shown to produce good quality, feasible routes for multiple agents.

However, it is also clear that substantially more research is still required to strip away the levels of abstraction and remove the naive assumptions before any such system could realistically see real-world deployment. As mentioned previously, any system seeking real-world use would require substantial government backing, likely requiring a mandated standard protocol and level of autonomous capability to be present in all new cars for a large amount of time, such that the vast majority of road-legal vehicles adhere to it, otherwise the system would need to be extended to plan around human-guided vehicles.

In my opinion, the assumption that all vehicles are fully-autonomous is not so much an assumption as it is a functional requirement for any system such as this to be efficient and achieve the net reduction in travel time that it attempts to.

The performance of the final incarnation of my parallel cooperative genetic algorithm is promising. Seeing substantial improvements in runtime, reliability and quality of generated routes over the course of its relatively short development period. I am confident that given more time it could be improved further, perhaps through persuing the distributed computation approach or parallelising further using GPU processing as mentioned in Chapter 5.

The implementation and evaluation of more complex genetic operators may also yield improvement, allowing the algorithm to explore the search space more thoroughly in fewer generations or, to better avoid generating infeasible routes, increasing the amount of feasible space a single individual can explore.

My macro level planner proved much more difficult to implement than anticipated. The modelling of a complex road network, including junctions and travel speeds, is beyond the scope of this project and as such I feel I was forced to make too many broad and naive assumptions which reduced the usefulness of the planner as a tool for evaluating my PCGA in a wider setting. In the future, with further research done into these sub-problems, I

believe such a system is possible.

Bibliography

[1] All vehicles (VEH01). https://www.gov.uk/government/statistical-data-sets/all-vehicles-veh01.

[2] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, Cambridge, UNITED STATES, 1992.

[3] Reported road casualties in Great Britain: Provisional estimates year ending June 2019. page 9.

[4] Rahul Kala. *On-Road Intelligent Vehicles: Motion Planning for Intelligent Transportation Systems / Rahul Kala.* Butterworth-Heinemann is an imprint of Elsevier, Kidlington, Oxford, UK, 2016.

[5] SN Bernstein. On the best approximation of continuous functions by polynomials of a given degree. *Comm. Soc. Math. Kharkow, Ser*, 2(13):49–194, 1912.

[6] Jens Gravesen. Adaptive subdivision and the length and energy of Bézier curves. *Computational Geometry*, 8(1):13–31, June 1997.

[7] G. Thippa Reddy, M. Praveen Kumar Reddy, Kuruva Lakshmanna, Dharmendra Singh Rajput, Rajesh Kaluri, and Gautam Srivastava. Hybrid genetic algorithm and a fuzzy logic classifier for heart disease diagnosis. *Evolutionary Intelligence*, 13(2):185–196, June 2020.

[8] Mahdi Shariati, Mohammad Saeed Mafipour, Peyman Mehrabi, Masoud Ahmadi, Karzan Wakil, Nguyen Thoi Trung, and Ali Toghroli. Prediction of concrete strength in presence of furnace slag and fly ash using Hybrid ANN-GA (Artificial Neural Network-Genetic Algorithm). *Smart Structures and Systems*, 25(2):183–195, 2020.

[9] Zixing Cai and Zhihong Peng. Cooperative Coevolutionary Adaptive Genetic Algorithm in Path Planning of Cooperative Multi-Mobile Robot Systems. *Journal of Intelligent and Robotic Systems*, 33(1):61–71, January 2002.

[10] A. Elshamli, H. A. Abdullah, and S. Areibi. Genetic algorithm for dynamic path planning. In *Canadian Conference on Electrical and Computer Engineering 2004 (IEEE Cat. No.04CH37513)*, volume 2, pages 677–680 Vol.2, May 2004.

[11] Rahul Kala. Optimization-Based Planning. In *On-Road Intelligent Vehicles*, pages 109–150. Elsevier, 2016.

[12] Luis Cruz-Piris, Miguel A. Lopez-Carmona, and Ivan Marsa-Maestre. Automated Optimization of Intersections Using a Genetic Algorithm. *IEEE Access*, 7:15452–15468, 2019.

[13] V. Roberge, M. Tarbouchi, and G. Labonté. Fast Genetic Algorithm Path Planner for Fixed-Wing Military UAV Using GPU. *IEEE Transactions on Aerospace and Electronic Systems*, 54(5):2105–2117, October 2018.

[14] Stephen A. Vavasis. Quadratic programming is in NP. *Information Processing Letters*, 36(2):73–77, 1990.

[15] Ardalan Vahidi and Antonio Sciarretta. Energy saving potentials of connected and automated vehicles. *Transportation Research Part C: Emerging Technologies*, 95:822–843, October 2018.

[16] Austin Brown, Jeffrey Gonder, and Brittany Repac. An Analysis of Possible Energy Impacts of Automated Vehicles. In Gereon Meyer and Sven Beiker, editors, *Road Vehicle Automation*, Lecture Notes in Mobility, pages 137–153. Springer International Publishing, Cham, 2014.

[17] Zia Wadud, Don MacKenzie, and Paul Leiby. Help or hindrance? The travel, energy and carbon impacts of highly automated vehicles. *Transportation Research Part A: Policy and Practice*, 86:1–18, April 2016.

[18] Chee Yap. Complete subdivision algorithms, I: Intersection of Bezier curves. In *Proceedings of the Annual Symposium on Computational Geometry*, volume 2006, pages 217–226, January 2006.

[19] Danny Hermes. Helper for bézier curves, triangles, and higher order objects. *The Journal of Open Source Software*, 2(16):267, August 2017.

[20] JuliaGraphics/Luxor.jl. JuliaGraphics, April 2021.

[21] JuliaAttic/Graphs.jl. JuliaAttic, March 2021.

[22] Seth Bromberger, James Fairbanks, and other contributors. JuliaGraphs/LightGraphs.jl: An optimized graphs package for the Julia programming language. 2017.

[23] JuliaGraphs/SimpleWeightedGraphs.jl. JuliaGraphs, March 2021.

[24] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[25] Sven Maerivoet and Bart De Moor. Cellular automata models of road traffic. *Physics Reports*, 419(1):1–64, November 2005.

[26] The Julia Programming Language. https://julialang.org/.

[27] C. Chen, Y. He, C. Bu, J. Han, and X. Zhang. Quartic Bézier curve based trajectory generation for autonomous vehicles with curvature and velocity constraints. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6108–6113, May 2014.

[28] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

Appendices

A Runtime Data

Procedure	ncalls	time	%tot	avg	alloc	%tot	avg
PCGA	1	14.4s	100%	14.4s	16.1GiB	100%	16.1GiB
Per-agent planning	3	14.4s	100%	4.81s	16.1GiB	100%	5.38GiB
fitness	3	14.4s	100%	4.81s	16.1GiB	100%	5.38GiB
base_fitness	135	7.55s	52.3%	55.9ms	5.65GiB	35.0%	42.9MiB
infeasible_distance	135	4.41s	30.6%	32.7ms	3.00GiB	18.6%	22.7MiB
high_prox_distance	135	3.14s	21.7%	23.2ms	2.66GiB	16.4%	20.1MiB
bezlength	135	724μs	0.01%	5.36μs	246KiB	0.00%	1.82KiB
collisionDetection	277	6.86s	47.6%	24.8ms	10.5GiB	65.0%	38.8MiB
bezInt	277	5.85s	40.5%	21.1ms	9.66GiB	59.8%	35.7MiB
mutation	3	1.71ms	0.01%	571μs	169KiB	0.00%	56.3KiB
crossover	3	1.14ms	0.01%	381μs	189KiB	0.00%	62.9KiB
genPop	3	406μs	0.00%	135μs	148KiB	0.00%	49.2KiB
update shared array	3	116μs	0.00%	38.5μs	8.31KiB	0.00%	2.77KiB
repair	3	98.1μs	0.00%	32.7μs	12.0KiB	0.00%	3.99KiB
filter isvalid	3	80.2μs	0.00%	26.7μs	45.9KiB	0.00%	15.3KiB
selection	3	75.1μs	0.00%	25.0μs	134KiB	0.00%	44.8KiB
sort by fitness	3	55.6μs	0.00%	18.5μs	2.25KiB	0.00%	768B

Table A.1: Runtime for a single generation over 3 sets of 15 individuals with intersection caching on, using Gaussian mutation, simple crossover and ranked selection.